O'REILLY®

TURING 图灵程序设计丛书



Hadoop应用架构

Hadoop Application Architectures

偏重Hadoop实践，直击企业大数据管理痛点，全面解析应用构架
阐述如何有效集成MapReduce、Spark、Hive等工具以形成完整数据解决方案

[美] Mark Grover，Ted Malaska，
Jonathan Seidman，Gwen Shapira 著
郭文超 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

郭文超

毕业于哈尔滨工业大学，浸淫开源社区多年，拥有丰富的平台建设及应用开发经验，善于解决各种疑难问题。现供职于神马搜索，负责商务搜索的离线架构方向。



图灵程序设计丛书

Hadoop应用架构

Hadoop Application Architectures

[美] Mark Grover, Ted Malaska, Jonathan Seidman, Gwen Shapira 著
郭文超 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

Hadoop应用架构 / (美) 马克•格洛沃
(Mark Grover) 等著 ; 郭文超译. -- 北京 : 人民邮电
出版社, 2017.1
（图灵程序设计丛书）
ISBN 978-7-115-44243-7

Ⅰ. ①H… Ⅱ. ①马… ②郭… Ⅲ. ①数据处理软件
Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2016)第292044号

内 容 提 要

本书讲解使用 Hadoop 平台进行应用架构所需要的关键知识，旨在帮助读者掌握有效集成 HBase、Kafka、Spark 等 Hadoop 生态圈工具以形成完整的大数据解决方案。书中内容分为两部分，第一部分介绍使用 Hadoop 创建应用程序时要考虑的问题，第二部分展示如何使用前面介绍的组件实现基于 Hadoop 的完整解决方案。

本书适合软件开发人员、构架师、项目主管等。

◆ 著　　[美] Mark Grover，Ted Malaska，
　　　　Jonathan Seidman，Gwen Shapira
译　　郭文超
责任编辑　朱 巍
执行编辑　张 憬
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：19
字数：449千字　　　　2017年 1 月第 1 版
印数：1－4 000册　　　　2017年 1 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-7590号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　案例研究

译者序

背景

时至今日，Hadoop 已形成了较为成熟、持续发展的生态圈。2016 年是 Hadoop 发展的第十个年头，从 v1 到 v2，再到将要发布的 3.0.0-GA，其功能、性能、稳定性及可用性均得到了极大的提升。Hadoop 在业界和学术界快速地渗透、迭代和普及，已经成为了数据处理领域最为基础的技术选型和基本架构。承担底层分布式存储的 HDFS、经典的分布式计算模型 MapReduce，以及成熟的资源管理任务调度框架 YARN 一同构成了传统概念上的 Hadoop。而基于 Hadoop 的各个组件也在蓬勃发展：进行内存迭代计算的新一代燎原之火 Spark 已至 2.0.2（2016 年 11 月 14 日），能将 SQL 转化成多种执行引擎（MapReduce、Tez、Spark）的 Hive 已至 2.1.0（2016 年 6 月 20），提供键值对存取的多版本海量数据库 HBase 已至 1.2.4（2016 年 11 月 7 日）。Hadoop 生态圈日趋庞大，各种各样的自由软件已逾百种（参见 http://hadoopecosystemtable.github.io）。

Hadoop 凭借开源社区的贡献者和布道师打造分布式环境下的大数据生态。然而，在大批软件不断涌现的背景下，也有一些曾风光无限或崭露头角的项目销声匿迹。开源社区中最为成功的产品非 Linux 莫属。在灵魂人物 Linus Torvalds 的带领下，Linux 进入服务器及嵌入式设备等领域，占领了操作系统的天下，又和 Android 一起占据了移动市场的大半江山，在桌面市场也有所斩获。众多 Linux 发行版大都将常用软件打包在内，并维护有自己的软件仓库，通过版本升级迭代增加新的功能、修复旧的 Bug，这些技术支持和服务都大大提高了 Linux 的易用性。与开源前辈 Linux 类似，Doug Cutting 大牛开创的 Hadoop 运行在廉价商用服务器上，以集群之力，分而治之地解决先前传统数据库、传统存储、传统计算模型束手无策的问题，让大规模数据的处理成为了可能。而 Hadoop 很早就进入了发行版时代，国外的 Cloudera、Hortonworks、DataBricks、MapR、EMC 等公司及国内的华为与星环都推出了自己的定制发行版本，各具特色地打包和修改 Hadoop 生态系统的软件、提供 BugFix 和 Backport 的 Patch，以及对应的增值服务和技术支持，如耳熟能详的 CDH、HDP 等。Amazon 的 EMR、阿里云的 MaxCompute（原 ODPS）、百度的 BMR 也是类似的产品。开源社区与企业有着不尽相同的业务场景和技术路线，单就交互式 SQL 引擎来讲，Cloudera 力推的 Impala 及优化存储 Kudu、Hortonworks 持续优化的 Hive、Spark 生态中原

生的 Spark SQL、Facebook 的 Presto、EMC 的 HAWQ、MapR 的 Drill 以及基于 HBase 的 Kylin 和 Phoenix 都是 SQL-on-Hadoop，但各有千秋。Hadoop 生态呈现碎片化趋势的同时，也有了百家争鸣的氛围。而开源与企业的结合，让 Hadoop 生态良性发展，也让数据唾手可得。作为开发者和用户，想要了解、使用、融入开源，除了各种博客、论坛、会议以及邮件列表之外，阅读文档和代码是不二之选。

关于本书

本书系统地展现了 Hadoop 生态圈的全景图，能够在面向问题解决的各种博客、论坛以及邮件列表之外，让读者可以在了解了整体架构和基本原理之后更好地去应用和实施，是一本面向体系建设和应用实现的教科书。所谓磨刀不误砍柴工，面向问题解决的思路如同充饥果腹的快餐，常常会急匆匆地解决问题，或不知其所以然、或不小心埋下深坑、或错失了更好的方案；而构建在基础知识和理论之上的架构体系和应用经验则是均衡营养、合理膳食的大餐，带来健康完善的思路、可以少走弯路、规避风险。更可贵的是，本书的第二部分专门介绍了大数据领域常见的三类应用场景，相信可以提高读者拆解业务需求、进行技术选型、更好地实现应用的能力。

致谢

感谢本书的编辑朱巍、岳新欣、谢婷婷、张憬，不辞辛苦地修改我扭曲的文字，让它可见天日。

感谢影响着我人生观价值观的祖父母及父母，让我可以安稳地长大成人，愿他们一世安详、永世安康；感谢我的妻子长久以来的支持和付出，愿我们分享阳光、分担风雨，一同携手走下去。

欢迎大家搜索“Hadoop 应用架构”QQ 群或直接输入“345527351”群号，加入本书的读者 QQ 群，交流大数据的那些事儿，解决大数据相关的各种问题。

是为序。

郭文超

2016年11月20日于北京

序

在过去的十年中，Apache 软件基金会的 Hadoop 项目蓬勃发展、欣欣向荣。

Hadoop 起初是 Nutch 项目的一部分，旨在提供一种前景远大的功能——通过扩展的方式支持到 PB 级数据的处理。2005 年，Hadoop 最多只能在几十台机器上运行，并且很不完善。只有一小部分人拿 Hadoop 做做试验，练练手。然而，一些人看到了它的前景与趋势，一种经济的、可扩展的、通用的数据存储和处理框架，有着广阔的实用价值。

到 2007 年，Hadoop 的高扩展性在 Yahoo! 公司得到了证实。目前 Hadoop 已经可以在数千台机器上可靠地运行了。Yahoo! 是第一个将 Hadoop 用于产品级应用的公司，而后其他互联网公司也相继使用，如 Facebook、LinkedIn 以及 Twitter 等。虽然这个时期的 Hadoop 在 PB 级的数据处理上拥有良好的扩展性，但考虑到无安全控制和只有 Java 语言的批处理接口，真正使用的话成本较高。

再后来，Hadoop 作为一个复杂生态系统的核心，增加了细粒度的安全控制、组件的高可用性（High Available，HA）以及通用的调度器（YARN，Yet Another Resource Negotiator，另一种资源协调者）。

围绕 Hadoop 这一核心，产生了一大批不同类型的工具。拿 HBase 与 Accumulo 来说，它们都能够提供在线的键值对存储，快速响应交互式的应用访问。而其他一些工具，如 Flume、Sqoop 以及 Apache Kafka，能够帮助完成数据在 Hadoop 存储层上的接入与导出。而 Pig、Crunch 以及 Cascading 提供了增强版的 API。SQL 查询能够通过 Apache Hive 与 Cloudera Impala 处理。Apache Spark 算是一个超级明星，能够提供更强大、更优化的批处理 API，同时还支持实时流处理、图数据处理与机器学习。Apache Oozie 与 Azkaban 能够协调和调度以上工具。

是不是对这些感到有些迷惑呢？叩开 Hadoop 生态系统的大门，一大波工具正汹涌地扑面而来。要想高效利用这一新平台，需要理解这些工具之间是如何适配的，以及哪个对你有所帮助。本书作者在基于 Hadoop 构建应用系统方面均拥有多年经验，本书就是这些经验和智慧的结晶。

理论上来说，有各种各样的方式配置和连接这些工具。但是实际上，工具的使用存在着成功的模式。本书描述了最好的实践经验，讲述每个工具的闪光点，以及怎样才能针对特定的任务发挥该工具的最大作用。另外，本书也提供了一些常见的用例。最初的使用者都没多少经验，尝试过各种工具的整合，但是本书描述了已被反复证明有效的模式，可以为读者节省大量的探索时间。

本书作者提供了使用这一强大平台所需要的基础知识。享受这本书吧，让它帮助你创建优秀的 Hadoop 应用！

Doug Cutting

加州院内棚中

前言

毫不夸张地说，在数据管理和数据处理领域，Apache Hadoop 带来了革命性的进展。Hadoop 的技术能力，使得许多行业中的组织能够解决以往技术不可能解决的问题。这些能力包括：

- 大规模数据的高扩展性处理；
- 不管什么格式和结构（或者缺少结构）的数据，都可灵活地处理。

Hadoop 另外一个值得注意的特点在于，它是一个意在相对廉价的商用硬件上运行的开源项目。相对于传统的数据管理解决方案来讲，Hadoop 在可以接受的成本范围内，提供了高扩展性和灵活性。

强大的技术能力，加上较低的经济成本，使得 Hadoop 及其生态系统中的诸多工具快速发展。而且，活跃的 Hadoop 社区也引入了大量支持 Hadoop 数据管理和处理的工具和组件。

尽管发展迅速，Hadoop 仍是一项相对年轻的技术。许多组织仍在尝试了解如何使用 Hadoop 来解决问题，以及如何将 Hadoop 及相关工具应用到真实场景中来形成解决方案。Hadoop 生态系统包含许多工具、应用编程接口（Application Programming Interface，API）及开发选项，这为开发人员提供了更多的选择余地和更大的灵活性，但也使得选择最佳的工具来实现数据处理应用成为了一项挑战。

我们在与大量客户协作的过程中，在与想要了解如何构建可靠、高扩展的 Hadoop 应用的 Hadoop 用户交流的过程中，积累了一些经验，受此启发编写了本书。本书目标并非为现存的工具提供详尽的文档描述，而是在基于 Hadoop 使用这些工具建设可扩展和可维护的应用架构方面，提供指引。

我们假定本书读者对 Hadoop 及相关工具有一定的经验。读者应熟悉 Hadoop 的核心组件，如 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）及 MapReduce。关于 Hadoop 及其核心概念，可参见 Tom White 的《Hadoop 权威指南》，这本书的确文如其名。

下面介绍本书中涉及的一系列比较重要的工具和技术，包括扩展阅读的参考资料。

- YARN

  直到不久前，Hadoop 的核心组件通常被认为是 HDFS 和 MapReduce。随着 Hadoop 中一个处理框架的引入，这种情况迅速发生了改变。由于 YARN 的引入，Hadoop 快速转型成为一个支持多种并行处理模型的大数据平台。YARN 为 Hadoop 数据处理提供了通用的资源管理器和调度器，不仅包括 MapReduce，还支持其他的数据处理模型。这使得在单个 Hadoop 集群上可以支持多个处理框架和多样的工作负载，并使得这些不同的模型和负载可以有效地共享资源。关于 YARN，欲了解更多，可参见《Hadoop 权威指南》或 Apache YARN 官方文档（http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html）。

- Java

  Hadoop 和与之相关的许多工具都是使用 Java 语言编写的，并且许多基于 Hadoop 的应用开发也是使用 Java 语言。虽然面向非 Java 开发者的工具和概念不断涌现，但是对于使用 Hadoop 的用户来讲，了解 Java 仍然是弥足轻重的。

- SQL

  虽然 Hadoop 为数据打开了多种处理框架之门，但 SQL 仍然是 Hadoop 数据查询的常用接口。这是因为大量的开发人员和分析人员熟悉 SQL，所以使用 Hadoop 时了解如何写 SQL 仍然是很有意义的。关于 SQL 的介绍，可以参考 Lynn Beighley 编写的 *Head First SQL*。

- Scala

  Scala 是一种在 Java 虚拟机（Java Virtual Machine，JVM）上运行的编程语言，它支持面向对象编程和函数式编程模型。虽然 Scala 是面向通用场景的编程语言，但是它已成为大数据领域越来越流行的语言了。无论是在与 Hadoop 交互的项目实现中，还是处理数据的应用开发中，皆是如此。使用 Scala 作为基础实现语言的项目如 Apache Spark 和 Apache Kafka。因此，基于 Spark 的应用开发也支持使用 Scala。在本书中，许多示例也是使用 Scala 编写的。如果需要了解 Scala，可参见 Cay S. Horstmann 所著的《快学 Scala》；若想深入了解，请参考 Dean Wampler 和 Alex Payne 合著的《Scala 程序设计（第 2 版）》[1]。

- Apache Hive

  提到 SQL，不得不提 Hive，它是一个用于 Hadoop 数据处理和数据建模的非常流行的工具，提供 HDFS 上数据的结构化定义，及数据的类 SQL 查询功能。Hive 项目中包含有一个元数据存储，它不仅以 Hive 的数据结构来存储元数据信息（就是描述数据的数据），还可供其他组件访问，如 Apache Pig（一个更高一层的并行编程抽象）和 MapReduce，其中后者需要借助 HCatalog 组件。另外，其他开源项目——如 Cloudera Impala，一个 Hadoop 之上的低延迟查询引擎——也可以使用 Hive 的元数据存储服务，该服务可以提供对 Hive 中预先定义的对象的访问。关于 Hive，欲了解更多，请参见 Hive 网站（https://hive.apache.org/）、《Hadoop 权威指南》，或 Edward Capriolo 等人所著的《Hive 编程指南》。

注 1：本书已由人民邮电出版社出版。——编者注

- Apache HBase

  HBase 是 Hadoop 生态圈中另外一个频繁使用的组件。它是一个分布式 NoSQL 数据存储，提供 HDFS 上超大规模数据集的随机访问。虽然被称为 Hadoop 数据库，但 HBase 与关系型数据库截然不同，熟悉传统关系型数据库系统的人要想了解 HBase，需接受新的概念。HBase 是许多 Hadoop 架构中的一个核心组件，本书中多有涉及。欲了解更多有关 HBase 的内容，可参考 HBase 网站（https://hbase.apache.org/）、Edward Capriolo 所著的《HBase 权威指南》，或 Nick Dimiduk 和 Amandeep Khurana 合著的《HBase 实战》。

- Apache Flume

  Flume 是一个常用的数据采集工具，将基于事件的数据（如日志）转存至 Hadoop。我们就 Flume 的最佳实践和部署架构进行了整体总结和细节描述。关于 Flume 的更多细节，可参见 Flume 文档（http://flume.apache.org/documentation.html），或《Flume：构建高可用、可扩展的海量日志采集系统》。

- Apache Sqoop

  Sqoop 是 Hadoop 生态圈中另外一个流行的工具，它用来在外部数据存储（如关系型数据库）与 Hadoop 之间进行数据移动。我们会讨论 Sqoop 的最佳实践，以及在 Hadoop 架构中它的最佳位置。关于 Sqoop 的更多细节，可参见 Sqoop 文档（http://sqoop.apache.org/docs/1.4.5/index.html），或 Apache Sqoop Cookbook（O'Reilly）。

- Apache ZooKeeper

  恰如其名的 ZooKeeper 项目旨在提供一个集中化的服务，用来保障 Hadoop 生态圈中各个项目间的协同工作。本书中提及的大量组件，如 HBase，就依赖于 ZooKeeper 提供的服务，所以对 Zookeeper 有基本的了解是有益处的。参考 ZooKeeper 网站（http://zookeeper.apache.org），或 Flavio Junqueira 和 Benjamin Reed 合著的《ZooKeeper：分布式过程协同技术详解》。

由此可见，本书的重点在于 Hadoop 生态圈中的开源工具。值得注意的是，很多传统企业级软件厂家提供了对 Hadoop 的支持，或者处于添加支持的过程中。如果你所在的公司已经使用了这样的企业级工具，那么尝试将这类工具集成到你的 Hadoop 应用开发环境中是大有裨益的。毕竟完成一项任务最好的工具是先前已经熟悉的工具。虽然了解本书中提及的工具，了解它们是如何集成到 Hadoop 中实现应用的，这些都是有意义的，不过在你的环境中选择使用第三方工具也是一个不错的选择。

重申一下，本书的目标不是介绍具体如何使用各种工具，而是讲述什么时候和为什么使用这样那样的工具，同时介绍最佳实践，以及最佳实践适用时的建议和不适用时的调整方法。我们希望本书能够对你构建成功的 Hadoop 解决方案有所帮助。

示例代码

就本书中的示例代码简单声明如下。我们尽量保证本书中的示例是最新的，并确保其正确性。获取最新版本示例代码，请访问本书的 GitHub 地址：https://github.com/hadooparchitecturebook/hadoop-arch-book。

目标读者

本书面向软件开发人员、架构师及项目主管等，满足大家了解 Apache Hadoop 及生态圈中工具的使用方法、建设端到端数据管理方案、集成 Hadoop 到已有数据管理架构等需求。我们的目标并不是深入研究特定的技术，比如 MapReduce，因为已有其他相关的参考资料。我们的目标是：介绍如何高效地集成 Hadoop 生态圈中的组件，以形成一个从原始数据开始直到数据消费掉的完整数据流水线，以及如何将 Hadoop 集成到已有的数据管理系统之中。

我们假定读者对 Hadoop 及相关工具（如 Flume、Sqoop、HBase、Pig 及 Hive 等）有所了解，但我们也提供了合适的参考资料作为补充内容。我们假定读者拥有 Java 编程经验，以及 SQL 和传统数据管理系统（如关系型数据库管理系统）的使用经验。

如果你是一名拥有 Hadoop 背景的技术专家，想要寻求架构或完整方案设计方面的最佳实践或者示例，那么本书再合适不过了。即使你是一名 Hadoop 专家，我们认为本书基于我们使用 Hadoop 的多年经验，包含了许多指引和最佳实践，仍然会让你有所受益。

通过本书，管理人员可基于实际的目标和项目情况，了解何种技术适用，从而为开发者选择合适的培训。

写作目的

多年以来，我们使用 Hadoop 构建大数据解决方案，无论是作为用户还是做客户支持，积累了一些经验。同时，Hadoop 市场也迅速成熟起来，关于深入了解 Hadoop 这一题材的资料也大量涌现。关于 Hadoop 及生态圈的相关工具，有大量的书籍、网站、课程等。尽管有如此多的资料，但在“有效集成这些工具以形成完整的解决方案”这一主题上，相关资源仍显不足。

与用户沟通时，无论这些用户是我们的客户、合作伙伴，还是与会人员，我们发现了一个共同的现象：在“对 Hadoop 有所了解”与“能够使用 Hadoop 解决实际问题”之间存在着巨大的鸿沟。举例来说，市面上有大量不错的资料可以帮助你了解 Apache Flume，但是如何判断这个工具是否适合你的用例呢？而且，一旦确定选择了 Flume 作为解决方案，怎样才能把它高效地集成到架构中呢？为了高效地使用 Flume，需要知道哪些最佳实践，以及需要做出哪些考量？

本书的目的就是缩小“对 Hadoop 有所了解”与“能够使用 Hadoop 形成实际解决方案”之间的差距。书中会介绍利用 Hadoop 实现解决方案时需要考虑的核心内容，并针对几个常见的用案提供完整的、端到端的解决方案示例。

本书结构

本书内容是按照在 Hadoop 上搭建解决方案的流程组织的，首先是 Hadoop 上的数据建模，接下来是将数据导入和导出 Hadoop，以及数据落地到 Hadoop 之后的数据处理，等等。当然，读者可以按照实际需求跳过部分内容。第一部分主要涵盖了使用 Hadoop 创建应用程

序时需要考虑的问题，包括以下几章。

- 第 1 章的内容是 Hadoop 中的数据存储和数据建模，例如文件格式、数据组织及元数据管理。
- 第 2 章的内容是 Hadoop 上的数据导入和导出。这一章将会讨论数据采集和抽取时需要考虑的问题和模式，包括常见工具的使用，如 Flume、Sqoop 和文件传输。
- 第 3 章介绍 Hadoop 上访问和处理数据的工具和模式。我们会在这一章讨论常见的数据处理框架，如 MapReduce、Spark、Hive 和 Impala，以及各自适合的应用场景。
- 第 4 章通过讲述 Hadoop 上一些常见用例的实现方案来继续讨论数据处理框架。我们会使用 Spark 和 SQL 实现具体的例子，来阐释如何解决常见问题，比如数据去重和时间序列数据的处理。
- 第 5 章主要讨论在 Hadoop 上处理海量图数据的工具，如 Giraph 和 GraphX。
- 第 6 章讨论将各种过程与应用协调和调度工具（如 Apache Oozie）整合在一起。
- 第 7 章讨论 Hadoop 上的近实时处理。我们在这里会讨论相对较新的流式数据处理工具，如 Apache Storm 和 Apache Spark Streaming。

在第二部分中，我们将会在 Hadoop 上端到端地实现一些常见的应用程序。这几章的目的在于提供翔实的案例，讲述如何使用第一部分中提到的各个组件，来实现一个基于 Hadoop 的完整解决方案。

- 第 8 章介绍了一个基于 Hadoop 的点击流分析的示例。对于运行大型网站的公司来讲，点击流数据的存储和处理是一个非常常见的用例。它也适用于处理任何机器数据的应用。这一章中，我们会讨论使用 Flume 和 Kafka 这样的工具进行数据采集，讨论如何高效地存储和组织数据，并展示处理数据的示例。
- 第 9 章介绍了一个基于 Hadoop 的欺诈检测应用的示例，这是 Hadoop 一个日益常用的应用场景。这一示例将会涵盖在欺诈检测的解决方案中如何使用 HBase，以及如何使用近实时处理。
- 第 10 章的案例研究探索的是另外一个常见用例：使用 Hadoop 扩展已有的企业级数据仓库（Enterprise Data Warehouse，EDW）环境。这包括将 Hadoop 作为 EDW 的补充，并提供传统数据仓库的基本功能。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。
- 等宽字体（`Constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`Constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*Constant width italic*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示、建议或一般注记。

该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/hadooparchitecturebook/hadoop-arch-book 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Hadoop Application Architectures* by Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira (O'Reilly). Copyright 2015 Jonathan Seidman, Gwen Shapira, Ted Malaska, and Mark Grover, 978-1-491-90008-6"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM

Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920033196.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我们非常感谢庞大的 Apache 社区，它们为 Hadoop 及其生态圈作出了很大的努力，否则本书将不可能存在。我们也非常感谢 Doug Cutting 先生为本书作序，更不用提他为 Hadoop 付出的一切。

有一大批人，他们的支持和努力使出版本书成为可能，首先是 Eric Sammer。Eric 一开始的支持和鼓励才使得本书成为现实。还有 Amandeep Khurana、Kathleen Ting、Patrick Angeles 以及 Joey Echeverria，他们都为本书早期的编写提供了珍贵的建议和反馈。

还有许多人在编写本书的过程中提供了珍贵的反馈和支持，尤其是那些花费时间和精力来审阅本书的人，他们是：Azhar Abubacker、Sean Allen、Ryan Blue、Ed Capriolo、Eric Driscoll、Lars George、Jeff Holoman、Robert Kanter、James Kinley、Alex Moundalexis、

Mac Noland、Sean Owen、Mike Percy、Joe Prosser、Jairam Ranganathan、Jun Rao、Hari Shreedharan、Jeff Shmain、Ronan Stokes、Daniel Templeton、Tom Wheeler。

Andre Araujo、Alex Ding 和 Michael Ernest 为代码示例的测试验证，慷慨地贡献了许多时间。Akshat Das 在绘图和网站方面提供了许多帮助。

有许多审稿人帮助我们审阅，极大地提高了本书的质量。如果仍存在错误，我们全权负责。

我们要感谢 Cloudera 的管理层，让我们得以编写本书。我们尤其要感谢 Mike Olson，感谢他持续不断的鼓励和支持。

我们要感谢 O'Reilly 出版社的编辑 Brian Anderson 以及制作编辑 Nicole Shelby，感谢他们在本书出版过程中的帮助与努力。另外，我们还要感谢 O'Reilly 内外的许多人——Ann Spencer、Courtney Nash、Rebecca Demarest、Rachel Monaghan 和 Ben Lorica，他们在本书不同的阶段帮助过我们。

最后，对没有一一列出的帮助过我们的朋友，我们表示歉意。

Mark Grover的致谢

首先我要感谢我的父母，Neelam 和 Parnesh Grover。是你们给了我每天如沐春风般的爱和支持。还要感谢我的妹妹 Tracy Grover，那个让我欺负、关爱、赞美的姑娘，感谢你的陪伴。感谢 Cloudera 公司的前主管 Arun Singla 和主管 Ashok Seetharaman，感谢你们对本书一如既往的支持。特别感谢 Paco Nathan 和 Ed Capriolo，是你们鼓励我写下了这本书。

Ted Malaska的致谢

我要感谢我的妻子 Karen，以及我最爱的两个讨厌鬼，TJ 和 Andrew。

Jonathan Seidman的致谢

我要感谢我生命中最重要的三个人，Tanya、Ariel 和 Madeleine，感谢他们在本书长久的写作过程中的耐心、关爱和支持。还要感谢本书的合著者，我的好伙伴 Mark、Gwen 和 Ted。最后，我把这本书献给已逝的父母，Aaron 和 Frances Seidman。

Gwen Shapira的致谢

我要感谢我的丈夫 Omer Shapira，感谢几个月来我写书时他给予我的精神上的支持和耐心。感谢我的父亲 Lior Shapira，也是我的金牌推销员，告诉他所有的朋友有这样一本"大数据书"。尤其是要感谢我的主管 Jarek Jarcec，感谢他的支持；感谢我的团队，大家去年为我分担了好多工作。

第一部分

考虑Hadoop应用的架构设计

第 1 章

Hadoop数据建模

本质上讲，Hadoop 是一种分布式数据存储，它为实现强大的并行处理框架提供了平台。该存储平台能在存储海量数据时保证高度的可靠性，同时具有灵活性，可以支持多个处理框架的运行。因此，Hadoop 成为了数据中心的绝佳选择。有了这样的特性，Hadoop 能够让你按原样存储任何一种类型的数据，对于数据的处理方式没有任何限制。

提到 Hadoop，我们常听到 Schema-on-Read 这个词。简单来讲，它指的是：未经处理的原始数据直接加载到 Hadoop 中，应用程序在处理这些数据时，按照需求暴露数据的结构。

通常在传统数据管理系统中应用的 Schema-on-Write 则不同。这种类型的系统要求在载入数据之前定义好存储模式。于是，数据需要先做好分析、建模、转换、加载、测试等一系列工作，然后才可以访问。另外，如果中间出现了决策错误或者需求变更，就必须重新开始这一连串的工作。如果尚未充分理解数据的结构或应用的特点，用户可以通过灵活的 Schema-on-Read 模式将数据的可见时间提前，这一点难能可贵。

关系型数据库与数据仓库一般适合进行常用的查询及报表访问，并且仅针对价值较高的数据。然而，Hadoop 正越来越多地承担起这类工作，需要在海量数据集上进行查询操作时尤其如此。面对这样的数据量，传统系统不是经济成本过高，就是技术尚有欠缺。

存储所有原始数据是一项非常强大的功能，但数据导入 Hadoop 之前，还是需要慎重考虑多种因素，包括以下几点。

- *数据存储格式*

  Hadoop 支持多种文件格式与压缩格式。每一种都有独特的优势，各自适合特定的应用。Hadoop 提供了用于存储数据的 HDFS（Hadoop Distributed File System），不过还有另外一些基于 HDFS 的系统也很常用，如提供数据（随机）访问功能的 HBase 与提供数据（表模式）管理功能的 Hive。这些系统也应当考虑。

- *多租户*

  集群通常可以支持多个用户、多个组以及多种应用类型。如果计划使用 Hadoop 来存储和管理数据，那么需要在多租户（multitenancy）支持方面多加考虑。

- *模式设计*

  尽管 Hadoop 本质上不关注模式，但是仍然需要考虑 Hadoop 中数据的存储结构。这主要包括将数据加载至 HDFS 的目录结构，以及数据处理与数据分析的输出目录结构。如果将数据存储到 HBase 和 Hive 这类系统中，则需考虑对应的数据模式设计。

- *元数据管理*

  像所有数据管理系统一样，描述存储数据的元数据往往和数据本身并重。对于元数据管理的理解以及相关决策不可轻视。

本章将讨论这些内容。值得注意的是，这些考虑因素是基于 Hadoop 构建应用的基础，也正因如此，这部分内容安排在了最前面。

对于 Hadoop 在存储数据上的应用，还有一个重要的考虑因素超出了本书的范围，这就是数据安全及相关问题。这主要包括用户认证、细粒度的访问控制以及数据（传输中的数据和落地后的数据）加密。更多关于 Hadoop 安全性方面的讨论，请参阅 Ben Spivey 与 Joey Echeverria 合著的 *Hadoop Security*（http://shop.oreilly.com/product/0636920033332.do，O'Reilly）。

1.1 数据存储选型

基于 Hadoop 构建解决方案时，最基本的决策之一是确定如何在 Hadoop 中存储数据。Hadoop 没有所谓的标准数据存储格式。但是跟标准文件系统相同，Hadoop 能够以任意一种格式存储数据，存储为文本、二进制、图像或者其他格式都可以。为了在存储和处理上更加优化，Hadoop 为很多格式内置了支持选项。这意味着 Hadoop 用户对数据拥有全部的控制权，有大量的选项可以选择。这不仅适用于采集到的原始数据，也适用于数据处理过程中产生的中间数据，以及数据处理后的导出数据。这也就是说，要以最佳方式存储数据，你需要做出很多决定。关于 Hadoop 数据存储，需要考虑的主要因素有几下几点。

- *文件格式*

  Hadoop 支持多种面向数据存储的文件格式，包括纯文本和 Hadoop 特有的格式，如 SequenceFile。还有一些更加复杂但功能更丰富的格式可供选择，如 Avro 与 Parquet。不同的格式具有不同的优势。任何一种格式都有适合的应用或者数据源类型。另外，你也可以在 Hadoop 中创建自己的定制化文件格式。

- *压缩格式*

  通常来说，与文件类型的选择相比，压缩格式的选择比较简单，但这仍然是一个需要考虑的重要问题。Hadoop 上常用的压缩编解码格式具有不同的特点，比如，一些编解码格式压缩和解压的速度较快，但是压缩效果不好，而有些编解码格式能将文件压缩得更小，但是压缩和解压的时间都很长，这种情况下，CPU 的负担无疑更重。在 Hadoop 上存储数据时，要考虑的另一个重要因素是压缩后的数据是否支持分片。这个问题我们将在后面详细探讨。

- *数据存储系统*

  尽管 Hadoop 中的所有数据最终存储在 HDFS 上，但是仍然需要选择实际的存储管理器（storage manager），比如你可以选择 HBase，也可以直接用 HDFS 存储数据。另外，Hive 和 Impala 这样的工具能够为 Hadoop 中的数据定义额外的结构信息。

在讨论 Hadoop 数据存储选项之前，需要注意以下两点。

- 本章将讨论不同的存储选项，但是数据存储的最佳实践将在后续章节深入讨论。比如，讨论采集数据到 Hadoop 时，我们再深入探讨存储这类数据时需要考虑的问题。
- 尽管本书重点关注 Hadoop 的 HDFS 文件系统，但是我们也不会忽略 Hadoop 的其他文件系统。其中包括开源文件系统，比如 GlusterFS 和 Quantcast，以及商用 Isilon OneFS 和 NetApp。亚马逊 S3（Simple Storage System，简单存储系统）这样的云存储系统也开始广泛应用了。文件系统也可能成为 Hadoop 部署方案中另一个需要考虑的架构因素。不过，这应该与我们在这里讨论的问题不相冲突。

1.1.1　标准文件格式

我们首先探讨 Hadoop 中标准文件格式的存储。这里，标准文件格式可以指文本格式，也可以指二进制文件类型。前者包括逗号分隔值（Comma-Separated Value，CSV）和可扩展标记语言文本（ Extensible Markup Language，XML）格式，后者包括图像。一般来说，在 Hadoop 中最好使用特定的文件格式来存储数据，这一点随后再进行讨论。但是在某些应用场景下，你可能想要按照原始格式存储源数据。如前所述，Hadoop 最强大的一个功能就是可以存储任何一种格式的数据。原始数据格式能够在线访问，数据完全保真，这意味着即使需求变更，你也可以对数据执行新的处理和分析操作。随后我们将讨论在 Hadoop 中存储标准文件格式时，需要注意的问题。

1. 文本数据

Hadoop 非常常见的一个应用是日志（如网络日志和服务器日志）存储与分析。文本数据当然也可以是其他很多格式，包括 CSV 文件和邮件之类的非结构化数据。在 Hadoop 中存储文本数据时，主要是要考虑文件在系统中的组织形式，1.2 节将详细讨论这一点。另外，因为文本文件会非常快速地消耗 Hadoop 集群的存储空间，所以最好对其压缩。还有一点需要谨记：使用文本格式存储数据会因类型转换带来额外开销。比如，一个文本文件中存储了 1234，如果将其作为一个整数使用，那么读取时就要进行字符串到整数的转换，写入时也要做相反的转换。1234 存储为文本会比存储为一个整数占用更多的空间。如果进行很多类似的转换，并存储大量这样的数据，那么这种开销加起来就很大了。

数据的使用方式会影响压缩格式的选择。如果是为了存档，你可能会选择压缩率最高的压缩格式，如果数据需要使用 MapReduce 进行处理，那么你可能想选择一种支持分片的压缩格式。可分片格式允许 Hadoop 将文件分成不同分片进行处理，这对数据的高效并行处理至关重要。本章后面的内容将讨论压缩类型及压缩格式，也会讲解可分片（splittability）的概念。

还有一点需要注意，在许多（甚至是大多数）场景中，运用 SequenceFile、Avro 等容器格式都能带来益处，因此容器格式是文本等大部分文件类型的优选格式。此外，容器格式还

能够支持可分片的压缩。本章后面将介绍这些容器格式。

2. 结构化文本数据

XML 和 JSON（JavaScript Object Notation，基于 JavaScript 语言的轻量级的数据交换格式）这样的结构化格式是文本文件格式中比较特殊的一种。XML 和 JSON 文件很难分片，Hadoop 也没有为这类格式提供内置的 `InputFormat`。在 Hadoop 中，JSON 格式比 XML 更难处理，因为这里没有标记可以标识记录的开始与结束。处理这类格式时，你有以下两个选择。

- 使用类似于 Avro 的容器格式。将数据转化为 Avro 的内容，从而为数据存储与数据处理提供更加紧密、有效的方法。
- 使用处理 XML 或 JSON 文件的专用库。比如，说到 XML，Pig 的 PiggyBank 库（https://cwiki.apache.org/confluence/display/PIG/PiggyBank）中就有 XMLLoader 可供使用。而 JSON 则可以利用 Elephant Bird 项目（https://github.com/twitter/elephant-bird）提供的 `LzoJsonInputFormat`。想要详细了解这些格式的处理，请参阅 Alex Holmes 所著的《Hadoop 硬实战》，该书提供了一些使用 MapReduce 处理 XML 与 JSON 文件的例子。

3. 二进制数据

Hadoop 存储中最常见的源数据格式是文本，但 Hadoop 也可以处理二进制文件（比如图像）。对于 Hadoop 上的大多数应用场景来说，二进制文件的存储和处理最好使用 SequenceFile 之类的容器格式。如果二进制数据的每个分片大于 64 MB，则可以考虑直接使用该二进制数据自己的格式，无需使用容器格式。

1.1.2 Hadoop文件类型

为了配合 MapReduce 的应用，Hadoop 也开发了一些专用文件格式，包括基于文件的数据结构（如 SequenceFile），也包括序列化格式（如 Avro）和列式存储格式（如 RCFile 和 Parquet）。以上文件格式各自具有不同的优点与缺点，但是所有的文件都具有以下特点，对于基于 Hadoop 的应用来说，这些特点都很重要。

- *可分片压缩*

  这些文件格式都支持通用压缩算法，并且都支持数据的分片。1.1.5 节将继续讨论这一话题。请注意，在 Hadoop 上存储数据时，很重要的一个考虑因素就是存储的文件是否支持分片。如果支持，那么大文件就可以分片，并作为 MapReduce 和其他类型任务的输入。文件分片以实现多任务处理，这是并行化处理的基础一环，也是应用 Hadoop 数据本地性特征的关键所在。

- *与压缩编解码无关*

  所有编解码器都支持文件压缩，读数据时并不需要关心使用了哪种编解码器。这是因为这些文件格式的头部元数据信息中能够存储编解码器。

下面将讨论基于文件的数据结构化格式，随后再讨论序列化格式与列式存储格式。

基于文件的数据结构化格式

Hadoop 最常用的基于文件的格式是 SequenceFile 格式，也有其他基于文件的格式可以使

用，如 MapFile、SetFile、ArrayFile 和 BloomMapFile。这些格式都是专门为 MapReduce 设计的，能够高效集成各种形式的 MapReduce 任务，包括通过 Pig 和 Hive 运行的任务。这里要介绍的 SequenceFile 是执行 Hadoop 任务最常用的格式。对于其他格式更完整的讨论，请参阅《Hadoop 权威指南》。

SequenceFile 以二进制键值对的方式存储数据，支持三种记录存储方式。

- 无压缩

  无压缩 SequenceFile 的 I/O（Input/Output，输入 / 输出）效率较差，而且会比压缩格式占用更多的磁盘空间。所以在大多数情况下，相比于有压缩的记录存储方式，无压缩的 SequenceFile 没有任何优势。

- 记录级压缩

  该格式对加入到 SequenceFile 中的每一条记录进行压缩。

- 块级压缩

  这种压缩方式会等待数据达到指定数据块大小，而不是在添加每条记录的时候都进行压缩。与记录级压缩相比，块级压缩的 SequenceFile 拥有更高的压缩率。通常来说，SequenceFile 都要选择块级压缩。另外，这里所说的块与 HDFS 或文件系统数据块无关。块级压缩中的块指的是压缩在一个 HDFS 数据块内的一组（多条）数据记录。

不管是哪一种存储方式，每个 SequenceFile 均包含一个通用的头部格式，头部包含与文件有关的基本元数据信息，比如所用的压缩编解码器、键和值的类名信息、用户定义的元数据以及随机产生的同步标志（sync marker）。这个同步标志也写入了文件内容，以便在文件中进行随机定位，这也是支持可分片的关键。举例来说，对于块级压缩的 SequenceFile，每个数据块的前面会有一个同步标志。

Hadoop 生态系统向 SequenceFile 提供了良好的支持，但是 Hadoop 之外的支持极为有限。SequenceFile 的支持语言只有 Java 这一种。通常，SequenceFile 会作为小文件的容器使用。在 Hadoop 中存储大量的小文件会引发一些问题。第一，这会导致 NameNode 过度使用内存，因为 HDFS 中每个文件的元数据都会存在内存里。第二，来自小文件的大量数据处理起来需要很多数据处理子任务，这会导致处理资源过度消耗。Hadoop 针对大文件进行过优化，因此，将小文件打包到 SequenceFile 中，能够实现更加有效的存储和处理。关于 Hadoop 小文件的相关问题以及 SequenceFile 的解决方案，更完整的讨论请参阅《Hadoop 权威指南》。

图 1-1 展示了使用块级压缩的 SequenceFile 的文件分布。需要注意的是，每个数据模块前都包含了同步标志，这样一来，文件在读取时就能够找到数据块的边界了。

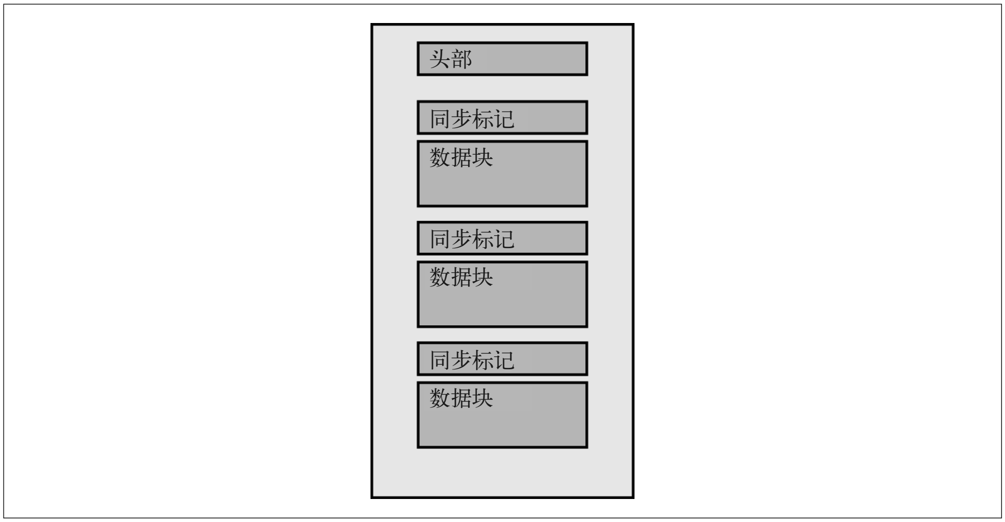


图 1-1：使用块级压缩的 SequenceFile 示例

1.1.3　序列化存储格式

序列化指的是将数据结构转化为字节流的过程，一般用于数据存储或网络传输。与之相反，反序列化是将字节流转化为数据结构的过程。序列化是分布式处理系统（比如 Hadoop）的核心，原因在于它能对数据进行转化，形成一种格式。使用了这样的格式之后，数据可以有效存储，也能通过网络连接进行传输。序列化通常与分布式系统中数据处理过程的两个方面紧密相连：进程间通信（比如远程过程调用，即 Remote Procedure Call，RPC），以及数据存储。考虑到本章的主题，我们将忽略 RPC，将本节的重点放在数据存储上。

Hadoop 主要采用的序列化格式为 Writables。Writables 的特点是紧密、快速，但是脱离 Java 语言便不易扩展和使用。不过，Hadoop 生态系统中也有越发普及的其他序列化框架，包括 Thrift、Protocol Buffers 与 Avro。其中，Avro 的适用性最好，因为它创建的初衷就是解除 Hadoop Writables 的限制。我们会详细介绍 Avro，但是在此之前，需要先认识一下 Thrift 和 Protocol Buffers。

1. Thrift

Thrift 是 Facebook 公司开发的框架，用于实现跨语言提供服务接口。Thrift 使用接口定义语言（Interface Definition Language，IDL）定义服务接口，而且依据 IDL 文件自动生成桩代码（stub code）。使用这些代码实现的 RPC 客户端与服务器，能够进行跨平台通信。Thrift 可以实现在多种语言中都能使用的接口，由此访问不同的后端系统。Thrift 的 RPC 层非常健壮，但是本章只关注作为序列化框架的 Thrift。尽管有时会用于 Hadoop 的数据序列化，但是 Thrift 仍然存在一些缺陷：不支持记录的内部压缩，不可分片，而且缺少 MapReduce 的原生支持。请注意，一些外部库（如 Elephant Bird 项目）能够消除这些缺陷，但是 Hadoop 没有为作为数据存储格式的 Thrift 提供原生支持。

2. Protocol Buffers

Protocol Buffers（protobuf）格式由 Google 公司开发，用于在不同语言编写的服务之间完成数据交换。与 Thrift 类似，Protobuf 的结构由一个 IDL 文件定义，IDL 用于为不同的语言创建桩代码。与 Thrift 类似的是，Protocol Buffers 不支持记录的内部压缩，不可分片，而且缺少 MapReduce 的原生支持。但是，同样与 Thrift 类似，Elephant Bird 项目可以用于编码 protobuf 记录，支持 MapReduce、压缩，以及分片。

3. Avro

Avro 是一种和语言无关的数据序列化系统，其设计初衷是解决 Hadoop Writables 的主要缺点，即缺少跨语言的可移植性支持。与 Thrift 和 Protocol Buffers 相同的是，Avro 的数据描述也无关语言。与 Thrift 和 Protocol Buffers 不同的是，Avro 可以选择生成代码，也可以选择不生成代码。因为 Avro 将模式存储于每个文件的头部，所以每个文件都是自描述的（self-documenting）。Avro 文件都很容易读取，即使是用一种语言写入数据，用另外一种语言来读取，也没有影响。Avro 为 MapReduce 提供了更好的原生支持，因为 Avro 的数据可压缩且可分片。Avro 的另一个重要特点是支持*模式演进*（schema evolution），这一特点使得 Avro 比 SequenceFile 更适合 Hadoop 应用。也就是说，读取文件的模式不需要与写入文件的模式严格匹配。于是，当需求发生变更时，模式中可以添加新的字段。

Avro 模式通常以 JSON 格式定义，但是也可以用 Avro IDL（一种类似于 C 的语言）定义。如前所述，模式存储于文件的头部，是文件元数据的一部分。除了元数据，文件头部还包括一个唯一的同步标志。与 SequenceFile 类似，这个同步标志用于隔开文件中的数据块，从而使 Avro 文件支持分片。每个 Avro 文件的头部后面都有一系列数据块，包含序列化后的 Avro 对象。这些数据块可以压缩。而且，各种数据以原格式存储在这些数据块中，这也为压缩提供了额外的帮助。本书撰写的时候，Avro 已向 Snappy 和 Deflate 格式的压缩提供了支持。

Avro 定义了少量的基本类型，包括 `Boolean`、`int`、`float` 和 `string`，它也支持 `array`、`map` 和 `enum` 等复杂类型。

1.1.4　列式存储格式

直到近几年，大多数数据库系统一直都以行存（row-oriented）的方式存储记录。需要获取记录中大多数列的时候，行存格式是较为高效的。举例来讲，如果进行分析的时候一定要获取某个时间段的全部字段记录，那么行存格式的存储就很实用。另外，行存格式会让数据写入更为高效，尤其是，写入时所有的记录行都已确定，在磁盘上定位一下即可。近几年，很多数据库引入了列式存储格式。与先前的行存系统相比，列式存储的系统能够提供以下优势。

- 对于查询内容之外的列，不必执行 I/O 和解压（若适用）操作。
- 非常适合仅访问小部分列的查询。如果访问的列很多，则行存格式更为适合。
- 相比由多行构成的数据块，列内的信息熵更低，所以从压缩角度来看，列式存储通常会非常高效。换句话说，同一列中的数据比行存数据块中的数据更为相似。当某一列的取值不多时，行存与列存格式在压缩效果上的差异尤为显著。

- 数据仓库类型的应用需要在极大的数据集上对某些特定的列进行聚合操作，而列式存储格式通常很适合此类应用场景。

显然，列式文件格式也常常出现在 Hadoop 应用中。Hadoop 支持的列式格式包括一度广泛应用为 Hive 格式的 RCFile，以及最近推出的其他格式，如 ORC（Optimized Row Columnar），以及 Parquet，后文中将详细介绍。

1. RCFile

RCFile 专为高效处理 MapReduce 应用程序而开发，尽管在实践过程中，它一般只作为 Hive 存储格式使用。RCFile 的开发旨在快速加载和查询数据，以及更加高效地利用存储空间。RCFile 格式将文件按行进行分片，每个分片按列存储。

与 SequenceFile 相比，RCFile 格式在查询与压缩性能方面有很多优势。但这种格式也存在一些缺陷，会阻碍查询时间和压缩空间的进一步优化。这些问题很多都可以由更为新型的列式存储格式（比如 ORC 与 Parquet）化解。大部分不断涌现的应用很有可能放弃使用 RCFile，改用新型的列存格式。不过，RCFile 目前仍然是 Hive 中的常用存储格式。

2. ORC

ORC 格式的开发初衷是为了弥补 RCFile 格式的一些不足，尤其是查询性能和存储效率方面的缺陷。相比 RCFile，ORC 格式在很多方面有显著进步，其特点和优势如下。

- 通过特定类型（type-specific）的 reader 与 writer 提供轻量级的、在线的（always-on）压缩。ORC 还支持使用 zlib、LZO 和 Snappy 压缩算法提供进一步的压缩。
- 能够将谓词下推至存储层，仅返回查询所需要的数据。
- 支持 Hive 类型的模型，包括新增的 decimal 类型与复杂类型。
- 支持分片。

撰写本书时，ORC 的劣势在于它是专门为 Hive 设计的，而不是通用的存储格式，无法用于非 Hive 的 MapReduce 接口（如 Pig 或 Java），也无法用于 Impala 这样的查询引擎。不过，有人正在研究如何弥补这个缺陷。

3. Parquet

Parquet 和 ORC 拥有很多相同的设计目标，但是 Parquet 有意成为 Hadoop 上的通用存储格式。实际上，ORC 出现的时间晚于 Parquet，所以有人会说 ORC 是 Parquet 的改进版。正因为如此，Parquet 的目标是成为能够普遍应用于不同 MapReduce 接口（如 Java、Hive 与 Pig）的格式，同时也要适应其他处理引擎（如 Impala 与 Spark）。Parquet 的优势如下，其中很多优势与 ORC 相同。

- 与 ORC 文件类似，Parquet 允许仅返回需要的数据字段，因此减少了 I/O，提升了性能。
- 提供高效的压缩，可以在每一列上指定压缩算法。
- 设计的初衷便是支持复杂的嵌套数据结构。
- 在文件尾部有完整的元数据信息存储，所以 Parquet 文件是自描述的。
- 完全支持通过 Avro 和 Thrift API 写入和读取。
- 使用可扩展的高效编码模式，比如，按位封装（bit-packaging）和游程编码（Run Length Encoding，RLE）。

Avro 与 Parquet 随着时间的推移，我们认识到，使用同一个接口读写 Hadoop 集群中的所有文件会带来很多益处。而且，如果要选择一种文件格式，那么最好选择支持模式描述的格式，因为 Hadoop 上的大部分数据最终都将成为结构化或者半结构化的数据。

既然需要模式的支持，那么 Avro 与 Parquet 就是很好的选择。不过，我们不想考虑是否需要创建一个模式的 Avro 版本与 Parquet 版本。好在这已经不是什么问题了，因为 Parquet 支持基于 Avro API 与 Avro 模式的读取和写入。

这样看来，万事俱备，我们可以使用同一个接口与 Avro 和 Parquet 文件交互，而且可以选择数据块存储和列式存储。

> **不同文件格式的失败行为**
>
> 不同文件格式之间一个重要的差异在于如何处理数据错误，某些格式可以更好地处理文件损坏。
>
> - 列式格式虽然高效，但是在错误处理方面表现并不是很好，这是因为文件损毁可能导致行不完全。
> - 序列化格式在第一个出错的行之前能够正常读取，但是在随后的行中无法恢复。
> - Avro 的错误处理能力最强，在出现错误记录时，读操作将在下一个同步点（sync point）继续，所以错误只会影响文件的一部分。

1.1.5 压缩

Hadoop 存储数据时需要着重考虑的另一个因素就是压缩。这里不仅要满足节省存储空间的需求，也要提升数据处理性能。在处理大量数据时，消耗最大的是磁盘和网络的 I/O，所以减少需要读取或者写入磁盘的数据量就能大大缩短整体处理时间。这包括数据源的压缩，也包括数据处理过程（如 MapReduce 任务）中产生的中间数据的压缩。尽管压缩会增加 CPU 负载，但是大多数情况下，I/O 上的节省仍然大于增加的 CPU 负载。

压缩能够极大地优化处理性能，但是 Hadoop 支持的压缩格式并不都是可分片的。MapReduce 框架先将数据分片，然后再输入多个任务，所以不支持分片的压缩格式对于数据的高效处理极为不利。如果文件不可分片，那就意味着需要将整个文件输入到一个单独的 MapReduce 任务，根本无法利用 Hadoop 提供的大规模并行以及数据本地化优势。因此，在选择压缩格式与文件格式时，是否支持分片是一个重要的考虑因素。我们将讨论可用于 Hadoop 的多种压缩格式，以及在选择不同格式时需要斟酌的一些因素。

1. Snappy

Snappy 是 Google 开发的一种压缩编解码器，用于实现高速压缩，适当兼顾压缩率。虽然压缩率不算突出，但是 Snappy 能够较好地平衡压缩速度和大小。Snappy 的处理性能显著优于其他压缩格式。值得注意的是，使用 Snappy 压缩的文件不是可分片的，所以它要与容器格式（如 SequenceFile 和 Avro）联合使用。

2. LZO

与 Snappy 相似，LZO 也具有较好的压缩速度，但压缩率略显平庸。与 Snappy 不同的是，使用 LZO 压缩的文件可分片，不过这里要求建立索引。如果纯文本文件不存储到容器格式中，那么使用 LZO 是一个不错的选择。有一点需要注意的是，LZO 的许可协议不允许将其打包到 Hadoop 中进行分发，因此需要单独安装。Snappy 则不同，它可以与 Hadoop 一起分发。

3. Gzip

Gzip 的压缩性能非常好，平均来讲，可以达到 Snappy 的 2.5 倍。但是它的写入速度不如 Snappy，平均为 Snappy 的一半。在读取性能上，Gzip 通常与 Snappy 相差不多。Gzip 同样是不可分片的，所以应该与容器格式联合使用。请注意，Gzip 处理数据有时会比 Snappy 慢，原因在于 Gzip 压缩文件需要的数据块较少，所以处理相同数据所需的任务就更少。因此，使用 Gzip 时选择较小的数据块，可以达到更好的性能。

4. bzip2

bzip2 的压缩性能很优越，但是处理性能明显比其他压缩编解码格式（如 Snappy）要差。与 Snappy 和 Gzip 不同，bzip2 本身为可分片式。在先前的例子中，从存储空间上看，bzip2 的压缩率比 Gzip 高约 9%。但是，额外的压缩要在读取、写入性能上付出很大的代价。在这个方面，不同的机器会有不同的性能差异，但是通常来说 bzip2 会比 Gzip 慢 10 倍。因此，bzip2 在 Hadoop 存储中并不是理想编解码格式，除非主要的需求就是减少存储空间占用量，例如在线归档时使用 Hadoop 的场景。

5. 压缩算法推荐

一般来说，在与容器文件格式（Avro、SequenceFile 等）一起使用时，任何压缩格式都可以是分片式的，因为容器文件格式能够单独压缩记录构成的数据块，也可以进行记录级的压缩。如果在压缩整个文件时没有使用容器文件格式，那么就需要使用本身支持可分片的压缩格式，比如在数据块之间插入同步标记的 bzip2。

以下是在 Hadoop 中进行压缩的一些建议。

- 开启 MapReduce 中间数据的输出压缩。这样可以减少需要读取和写入磁盘的中间数据，进而提高了性能。
- 注意数据是如何排序的。通常来讲，数据应当排序，相似的数据要放在一起，这样压缩率更高。需要记住的是，Hadoop 文件格式中的数据是以块为单位压缩的，而且那些块的熵将决定最终的压缩。比如，如果数据包括带有时间戳的股票基点、股票报价以及股票价格，那么相比按照含唯一值的列（如时间或股票价格）排序，按照重复的字段（如股票报价）排序压缩效果更好。
- 考虑使用支持可分片压缩算法的紧凑文件格式，如 Avro。图 1-2 展示了使用不可分片压缩算法的 Avro 或 SequenceFile 支持文件分片的方法。一个 HDFS 数据块能够包含多个 Avro 或 SequenceFile 数据块。每一个 Avro 或 SequenceFile 数据块能够独立于其他的 Avro 或 SequenceFiles 数据块，单独进行压缩和解压操作。这也保证了每一个 HDFS 数据块都能被单独压缩与解压，由此实现了数据可分片。

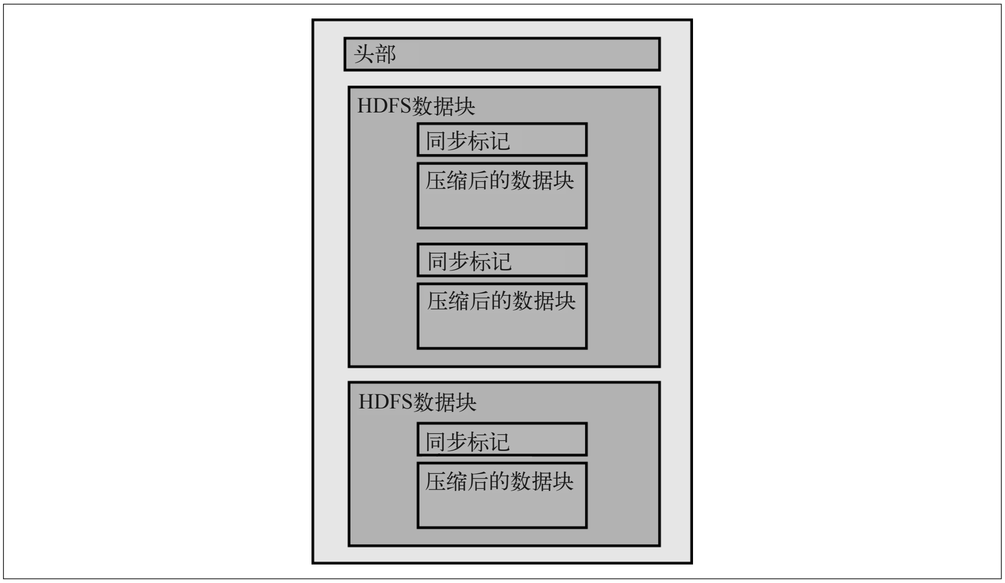


图 1-2：Avro 压缩示例

1.2 HDFS模式设计

如 1.1 节所述，HDFS 和 HBase 是两种很常用的存储管理器。你可根据实际情况选择用 HDFS 或 HBase（内部数据存储于 HDFS）存储数据。

本节将探讨哪一种模式更有利于将数据直接存储于 HDFS。如前面提到的，数据加载到 Hadoop 中时，Hadoop 的 Schema-On-Read 模型对数据没有任何要求。数据可以通过众多方法（将在第 2 章深入讨论）便捷地输入 HDFS 中，而不需要参考模式信息，也不必预处理数据。

尽管有很多人用 Hadoop 存储和处理非结构化数据（如图像、视频、邮件或者博文）和半结构化数据（如 XML 文档和日志文件），做些约定还是有必要的。Hadoop 经常充当整个组织的数据中心，HDFS 中存储的数据要在许多部门和团队之间共享。在创建数据库的时候仔细进行构建和组织能够带来很多益处，包括以下几条。

- 标准化的目录结构可以让不同的团队更加轻松地使用同一个数据集共享数据。
- 支持实施访问控制和配额控制，可以防止意外删除或损坏。
- 在所有数据都准备好进行处理之前，使用者时常会将数据存放在一个独立的区域。存放数据的约定有助于确保部分加载数据不会被当作加载完成的数据，从而被意外处理。
- 对数据进行标准化的组织，使得某些数据处理代码可以复用。
- Hadoop 生态系统中的一些工具有时会对数据的位置作出假设。当初次将数据加载到 Hadoop 中时，符合那些假设通常会容易些。

数据模型的细节和具体的用例高度相关。比如，数据仓库的实现和其他事件存储很可能采

用类似于传统星型的模型，包括结构化的事实表和维度表。另一方面，非结构化与半结构化数据可能将重点更多地放在目录位置与元数据管理方面。

设计模式时，无论项目需求如何，以下几点都要铭记在心。

- 要创建实践标准并遵守标准，特别是当多个小组共同使用数据时。
- 确保你的设计能与计划使用的工具配合使用。比如，计划使用的 Hive 版本可能只支持目录中以特定方式命名的表分区。这通常会影响模式设计，尤其会影响子目录表的命名方式。
- 设计模式时要记住使用模式。不同的数据处理与查询模式要配合不同的模式设计。敲定模式之前，你要理解主要用例和数据获取要求，从长远看，这可以保证模式更易维护和支持，也更易提高数据处理性能。

1.2.1　文件在HDFS中的位置

为了用更具体的术语来进行讨论，设计一种 HDFS 模式时，首先应该决定文件的位置。标准化的位置使团队之间更容易查找和共享数据。下面是我们推荐的 HDFS 目录结构示例。目录结构简化了不同组和用户的权限分配。

- /user/<username>

  只属于特定用户的数据、JAR 包和配置文件。这通常是用户在试验中使用的非正式数据，不属于业务流程。/user 下的目录通常只能由所有者进行读取和写入。

- /etl

  ETL（Extract, Transform and Load，提取、转化、加载）工作流正在处理的、处于不同阶段的数据。/etl 目录由 ETL 过程（通常在各自的 user 目录下进行）与 ETL 团队的成员读取和写入。拥有 ETL 过程的不同组别（如业务分析、欺诈识别以及市场营销）在 /etl 目录树中有对应的子目录。ETL 工作流通常隶属于大型应用，比如点击流分析或者推荐引擎，而且每一个应用在 /etl 目录下都要有对应的子目录。在每一个应用的目录中，该应用的每个 ETL 过程或工作流都有一个目录。工作流目录中，每个数据集都有一个子目录。比如，如果商业智能（Business Intelligence，BI）组有一款点击流分析应用，而且其中一个过程是聚合用户的喜好，那么建议将包含该数据的目录命名为 /etl/BI/clickstream/aggregate_preferences。有些情况下，使用者可能想让目录树更深一层，为过程的每个阶段都建立目录：数据着陆的目录为 input，中间处理阶段的目录为 processing（可能存在多个 processing 目录），最终结果的目录为 output，而被 ETL 过程拒绝、需要进行手动故障处理的数据或者文件则进入 bad 目录。这种情况中，最终的结构大约是这样的：/etl/<group>/<application>/<process>/{input、processing、output、bad}。

- /tmp

  工具生成或者用户共享的临时数据。该目录通常通过程序自动清除，不会存储生命周期长的数据。通常每个人都能读取或写入该目录。

- /data

  经过处理并且在整个组织内共享的数据集。这些通常是待分析数据的重要来源，可以

促成业务决策，所以不能不分身份角色，任人读取和写入。通常，用户只能读取数据，数据由自动化的 ETL 过程写入，而且要受到审计。/data 目录下的数据通常对于业务非常重要，所以一般只允许自动化的 ETL 过程写入数据，改变都是要受到控制和审计的。不同业务团队对于 /data 目录下的目录有着不同的读取权限，这取决于他们的报告与处理需求。在 /data 中存储的数据集应当经过了处理，用于共享，所以其中每一个数据集都有一个子目录。比如，如果将一种药物的所有订单存为一个名为 medication_orders 的表格，那么我们建议将该数据集存储于一个名为 /data/medication_orders 的目录中。

- /app

  几乎囊括 Hadoop 应用运行所需要的一切，但不包括数据。这里有 JAR 文件、Oozie 工作流定义、Hive HQL 文件，等等。应用的代码目录 /app 用于存储应用所需的依赖，如用于 Oozie 工作流的 JAR 包，以及 Hive 用户自定义函数（User-Defined Functions，UDF）的 JAR 包。一般来讲，HDFS 中不是必须存储应用程序的这些内容，但是一些 Hadoop 应用（比如 Oozie 与 Hive）要求将共享编码和配置存储到 HDFS 上。这样一来，在集群的任一节点执行的代码都可以使用它。该目录中每一个组和应用都应该包括一个子目录，类似于 /etl 的结构。对于一个给定的应用（比如 Oozie），如果使用者决定将应用程序存储到 HDFS 中，那么应用的每个版本都需要一个目录。有可能通过 HDFS 的符号链接做标记，最新的标为 latest，当前使用的标为 current。目录包含的二进制程序能够呈现在各个版本的目录中。类似于：/app/<组>/<应用>/<版本>/<包目录>/<包>。继续刚才的例子，聚合处理喜好，这个应用最新创建的 JAR 所在的目录结构类似于：/app/BI/clickstream/latest/aggregate_preferences/uber-aggregatepreferences.jar。

- /metadata

  存储元数据。尽管大多数表元数据都存储在 Hive metastore 中（即将在 1.4 节介绍），但是还有一些元数据（如 Avro 模式文件）可能需要存储在 HDFS 中。该目录是存储此类元数据的最佳位置。该目录通常对 ETL 任务可读，而采集数据到 Hadoop 中的用户（如 Sqoop 用户）则拥有写权限。比如，一个数据集的 Avro 模式文件被称作 `movie`，那么它可能位于 /metadata/movielens/movie/movie.avsc。第 10 章将详细讨论这一特定示例。

1.2.2 高级HDFS模式设计

一旦确定了大体的目录结构，接下来就要决定如何将数据组织到文件中。我们讨论过，数据加载形成的格式可能不是存储数据的最佳格式，但值得注意的是，数据加载的默认组织形式也不一定是最好的。我们可以采用一些策略，对数据作出最好的安排和组织。这里将讨论分区、分桶以及反向规范化。

1. 分区

对数据集进行分区是一种很常见的技术，目的是减少处理该数据集时所需要的 I/O。处理大量数据时，减少 I/O 带来的好处显而易见。但是，与传统数据仓库不同的是，HDFS 不会将索引存储到数据中。索引减少了，数据加载速度会有所提升，但是也意味着每次查询都将读取整个数据集，只处理一个小的数据子集（即全表扫描，full table scan）时也是如此。当数据集很大，而查询仅访问数据的子集时，一个很好的解决办法就是将数据集分为更小的子集或者分区。整个数据集对应一个目录，而每个分区都会在这个目录的子目录中

呈现。这就使得查询可以只读取所需的特定分区（比如子目录），因而减少了 I/O，也显著地提高了查询性能。

举个例子，你建立了一个存储订单的数据集，订单内容为各种药品，数据集的名字为 medication_orders。你想查看过去三个月中一个药剂师的订单。如果不分区，你就必须读取整个数据集，然后筛掉所有与查询不相符的记录。

但是，如果我们对整个订单数据集进行分区，让每个分区只包含一天的数据，那么查询过去三个月的数据就只需要读取大约 90 个分区，而不是整个数据集。

将数据存储于文件系统时，应该使用以下目录格式进行分区：< 数据集名称 >/< 分区列名 = 分区列值 >/{ 文件 }。在这个例子，分区写作：medication_orders/date=20131101/{order1.csv, order2.csv}。

包括 HCatalog、Hive、Impala 和 Pig 在内的很多工具都可以识别这种目录结构。这些工具可以利用分区减少处理数据集时所需要的 I/O。

2. 分桶

分桶是另外一种技术，能将大数据集分解为更容易处理的数据集。分桶与许多关系型数据库所使用的哈希分区相类似。在先前的例子中，我们可以将订单数据集按照日期进行分区。这是因为每天都有大量的订单，而分区所包含的文件足够大，HFDS 的优化得以体现。但是，如果我们试图按照药剂师对数据进行分区，以优化针对药剂师的查询，那么分区数量可能会特别多，而且分区文件可能会特别小。这就会导致小文件的问题（small files problem）。因为 HDFS 会将每个文件的元数据都存储于内存中，所以如 1.1.2 节中所讲的，在 Hadoop 中存储大量的小文件会导致 NameNode 内存的过量使用。同样，小文件越多，要处理的任务就越多，数据处理的压力就会非常大。

这个问题的解决方法就是对药剂师进行分桶，即使用哈希函数将药剂师映射到数量特定的桶。这样，数据子集（即桶）的大小就受到了控制，查询速度也得到了提升。文件不应该太小，否则就会有大量的小文件需要读取和管理。但是文件也不能太大，扫描大量数据会减慢查询速度。一般来说，桶的大小达到 HDFS 数据块大小的若干倍会比较合适。哈希到桶列时，数据的平均分布至关重要。这是因为平均分布可以保证桶的稳定性。通常，桶的数量是 2 的 N 次幂。

当关联两个数据集时，分桶的另一个好处就显而易见了。这里的关联指的就是一种很常见的概念：将两个数据集关联，得到一个结果。一些 SQL-on-Hadoop 系统能够进行以上所说的关联，MapReduce、Spark 或其他 Hadoop 的编程接口也可以完成关联。

如果关联的数据集恰好按照关联的键分桶，而且一个数据集中桶的数量是另一个的倍数，那么就足够单独关联相应的桶，而不需要关联整个数据集了。这显著降低了两个数据集执行 Reduce 端关联（Reduce-side join）的时间复杂度。这是因为 Reduce 端的关联非常消耗资源。但是，如果关联的是两个桶数据集，而不是两个整数据集，那么关联相应的桶即可。这样就可以减少关联损耗。当然，来自两个表的不同的桶可以并行关联。另外，分桶之后数据量通常都比较小，一般能够放入内存。所以整个关联操作可以在 Map-Reduce 任务的 Map 阶段通过将小桶加载到内存中进行。这就是所谓的 Map 端关联（Map-side join），与 Reduce 端关联相比，它的性能更好。如果使用 Hive 进行数据分析，应该能自动识别分

桶的表并执行这种优化。

如果桶中的数据是*有序的*（sorted），那么就可以使用合并连接（merge join），而且关联时不会将整个桶都存入内存。这比简单的桶关联（bucket Join）更快，而且更加节省内存。Hive 也支持这种优化。注意，任何一个表都可以分桶，在没有逻辑意义的分区点上也是如此。对于经常进行关联操作的大表，最好对数据进行排序和分桶，而且要按照关联字段分桶。

正如先前所讨论的，模式设计高度依赖数据查询的方式。在决定数据分区与分桶之前，你需要了解有哪些列要用于关联和过滤。如果存在多个普通查询模式，而且很难决定分区字段，那么可以选择多次存储同一数据，每次都采用不同的物理组织形式。这在关系型数据库中称作反模式设计，但是在 Hadoop 中，这种是一种可行的方案。首先，Hadoop 中的数据通常是一次性写入的，而且几乎不会进行更新。由此，同步保存复制的数据集带来的损耗通常会大大减少。另外，Hadoop 集群的存储成本也显著地降低了，所以更不用担心浪费磁盘空间。应用这些特性，我们可以用一部分空间换取更快的查询速度，这样做通常会带来让人满意的结果。

3. 反向规范化

先前的小节讨论了关联。除此之外，还有一种方法可以用磁盘空间换取查询性能，也就是将数据集反向规范化，削减关联数据集的需求。在关系型数据库中，数据通常以*第三范式*（third normal form）存储。该模式将数据分片成更小的表，使每个表都含有一个特定的实体，从而将冗余减到最少并且提供数据完整性。这意味着大多数查询为了得到最后的结果数据集，需要将大量的表关联在一起。

但是，在 Hadoop 中，关联通常是最慢的操作，而且消耗的集群资源最多。Reduce 端的关联尤其如此，因为它需要通过网络分发表的所有数据。就像我们已经看到的，重点在于通过优化模式尽量避免这些代价高昂的操作。分桶与排序在这里很实用，另外一种方法是让数据集预先连接，这就是预聚合。这里的思路是尽量提前进行查询，尤其是那些很有可能经常执行的查询或者子查询，预先将查询的工作量减到最少。用户不必每次查看数据时都运行关联操作，我们可以一次性关联数据，并将其存储起来。

针对特定的使用案例，观察典型的在线事务处理（Online Transaction Processing，OLTP）模式与 HDFS 模式之间的差异，你会发现，Hadoop 在 ETL 处理过程中将很多小维度表联合至更大的维度。在先前关于药品的示例中，为了避免重复关联，我们将频率、类别、管理途径以及单元关联到了药品数据集。

其他类型的数据预处理，如聚合或者数据类型转换，也都能加快处理速度。因为数据重复不是什么特别大的问题，所以大量查询中经常出现的处理类型都值得一次性处理，重复使用。在关系型数据库中，这种模式就是所谓的*物化视图*（Materialized Views）。在 Hadoop 中，这需要创建一个新的数据集，包含其聚合格式中的相同数据。

1.2.3 HDFS模式设计总结

我们在此概述一下重点。本节介绍了通过有选择地读写特定分区中的数据，使用分区技术减少数据处理所需的 I/O。我们还学习了如何使用分桶技术，减少 I/O，从而加快关联及采

样的查询速度。最后，我们明白了反向规范化在加快 Hadoop 任务处理速度方面起到的重要作用。我们已经对 HDFS 设计模式有了大致了解，下面将讨论 HBase 模式设计的相关概念。

1.3 HBase模式设计

本节将描述如何设计良好的 HBase 数据存储设计模式。HBase 是一个很复杂的话题，已经有很多书籍介绍了 HBase 的使用与优化。本章将从更高的层面介绍 HBase，主要关注那些通过 HBase 解决常见问题的通用设计模式。想了解 HBase，请参阅《HBase 权威指南》和《HBase 实战》。

首先需要理解的是，HBase 不是关系型数据库管理系统（Rational Database Management System，RDBMS）。实际上，为了更好地理解 HBase 的工作方式，最好把 HBase 当作一个巨大的哈希表。与哈希表相同的是，HBase 允许用户将指定的键和特定值进行关联，这样一来，基于给定的键就能快速查找到相应的值。

相比哈希表，HBase 在应用时涉及的细节更多，包括 Region 和 Compaction 的工作原理、向 HBase 导入数据的不同策略，以及数据块缓存的使用和理解。不过，HBase 仍然值得同哈希表进行类比，主要是因为 HBase 更多地被当作一种分布式键值存储，而不是 RDBMS。这样你会联想到 `get`、`put`、`scan`、`increment` 和 `delete` 等请求，而不是 SQL 查询。

在关注可以在 HBase 中完成的操作之前，我们先回顾一下哈希表支持的操作。

(1) 将数据放置于哈希表中。
(2) 从哈希表中获取数据。
(3) 迭代访问哈希表 [注意：HBase 通过*范围扫描*（range scan）提供了更为强大的迭代访问策略，能够在扫描时指定开始行和结束行]。
(4) 对哈希表中的一项增值。
(5) 删除哈希表中的值。

这样，放弃 SQL 而使用 HBase 的理由也一目了然了。HBase 的价值在于可扩展性和灵活性。HBase 在很多应用中都非常实用，常用于欺诈检测，第 9 章将详细讲解。总之，HBase 能够通过 `get` 与 `put` 请求解决问题。

我们已经对 HBase 有了大致的了解，也学习了背景知识，下面就从架构的角度讨论如何设计一个良好的 HBase 模式。

1.3.1 行键

继续与哈希表类比，HBase 中的行键类似于哈希表中的键。要构造一个良好的 HBase 模式，关键之一就是选择一个合适的行键。下面介绍行键在 HBase 中的具体应用方式，以及如何选择行键。

1. 记录检索

行键是在 HBase 中检索记录所使用的键。HBase 记录含有的列在数量上没有限制，但是只能有一个行键。这一点同关系型数据库有所不同，后者的主键可以由多个列组合构成。既

然要为记录创建唯一的行键，那么单一的行键就需要包含多种信息。例如，如果一行表示一个订单，那么 `customer_id`，`order_id`，`timestamp` 可以作为该行的行键。在关系型数据库中，`customer_id`、`order_id` 和 `timestamp` 是三个分开的列，而 HBase 中三者需要组合形成一个独特的标识符。

选择行键时需要记住的另外一点是，单一记录中的 `get` 操作是 HBase 中最快的操作。因此，在设计 HBase 模式时，用 `get` 操作解决大多数常见的数据使用问题，这样会提高处理性能。这可能意味着将大量数据放到单一记录中，比关系型数据库中存放的数据多。这样的设计被称作*反向规范化*，不同于关系型数据库中常见的*规范化设计*。比如，关系型数据库可能会将消费者存在一个表中，将其联系方式存在另一个表中，又将订单存在第三个表中，而订单详情存在另外的表中。HBase 则会设计非常宽的表，即每一个订单记录都包含该订单的所有信息，也包含消费者及其联系方式。单一的 `get` 操作可以检索所有的数据。

2. 分布

对于一个给定的表，记录在 HBase 集群多个 Region 中的分布方式是由行键决定的。HBase 中所有的行键都进行了排序，经过排序的行键按照范围存储在不同的 Region 中。每个 Region 都固定在一个 Region 服务器（即集群中的一个节点）上。

行键有一个著名的的反模式设计，那就是时间戳的使用。在行键中使用时间戳可以将大多数 `put` 和 `get` 请求集中在单一的 Region 上，进而集中在单一的 Region 服务器上。于是，在某种程度上，HBase 不再是一个分布式系统。一般来说，选择行键的最合理标准是让集群的负载分布均匀。本章后面将会谈到，解决该问题的一个方法就是对键进行 salt 处理。在机器数据的键 salt 处理上，设备 ID 与时间戳结合以及反转时间戳的使用尤为常见。

3. 数据块缓存

数据块缓存是一种最近最少使用的（Least Recently Used，LRU）缓存，能将数据块缓存到内存中。默认情况下，HBase 按照 64KB 的块大小从磁盘中读取记录，每一个这样的数据块都是一个 HBase 数据块（HBase block）。从磁盘中读取的时候，HBase 数据块会放置在数据块缓存中。但是，你也可以不将 HBase 数据块放到数据块缓存中。之所以要进行缓存，是因为最近访问的数据（以及与这些数据同在一个 HBase 数据块中的数据）很有可能在不久之后再次被请求访问。但是，由于数据块缓存的大小有所限制，所以应该谨慎明智地使用它。

选择不合适的行键会导致不理想的数据块缓存分布。比如，如果选择某个属性的哈希作为行键，那么 HBase 数据块中的那些记录很可能并不相关。这样一来，数据块缓存中将填充不相关的记录，导致缓存的命中率会非常低。在这种情况下，可以采用另外一种设计：对行键的第一个部分进行 salt 处理，让同一个 HBase 数据块中同时获取的数据之间按行键有序排列。salt 处理是对原来的键或键的一部分做哈希取模，所以可以单独地通过原始键生成。我们将在第 9 章举例讲解键的 salt 处理。

4. 支持扫描

选择合适的行键能够使相关的记录位于同一个 Region。这有助于进行范围扫描，因为 HBase 只需扫描一定数目的 Region 就可以获取结果。另外，如果行键选择不合适，范围扫描在获取数据时可能需要扫描多个 Region 服务器，随后还要过滤掉不相关的记录。这样一

来，同样的请求就增加了额外的 I/O。另外，要记得 HBase 的扫描速度大约是 HDFS 的八分之一，降低 I/O 要求能显著提高性能。与 HDFS 上的数据存储相比，HBase 在这一点上更为明显。

5. 行键大小

行键的大小将决定工作负载的性能。通常来说，行键越短越好，因为这样可以降低存储消耗，提高读写性能。但是，行键较长时，`get` 和 `scan` 的性能更好。因此，在选择时，需要权衡行键的长短。

下面看一个例子。表 1-1 为 HBase 中包含三条记录的表。

表1-1：HBase表举例

行键	时间戳	列	值
RowKeyA	Timestamp	ColumnA	ValueA
RowKeyB	Timestamp	ColumnB	ValueB
RowKeyC	Timestamp	ColumnC	ValueC

行键越长，压缩编解码在存储行键时需要处理的 I/O 就越多。同理，列的名称越长，需要处理的 I/O 也越多。所以一般来说，保持列的名称短一些比较好。

> 可以使用 Snappy 配置 HBase，压缩行键。因为行键按照排序进行存储，所以行键之间的排列比较紧密时，压缩效果较好。这就是不宜将某个属性的哈希作为行键的另一个原因，因为行键的排序是任意的。

6. 可读性

可读性是比较主观的概念。但是既然行键的应用非常常见，那么它的可读性就不容小觑。一般来说，你应当先接触可读性比较好的行键，如果你还不太习惯 HBase，那就更应该这样做了。这能让你更为轻松地发现和调试问题，也更容易使用 HBase 操作台。

7. 唯一性

因为行键等同于哈希表类比中的键，所以确保行键的唯一性非常重要。如果你选择了一个基于非独特属性的行键，那么应用应该处理这种情况，而且只能通过唯一的行键把数据存入 HBase。

1.3.2　时间戳

要获得良好的 HBase 模式设计，第二个要点是正确理解和使用时间戳。在 HBase 中，时间戳的作用如下所述。

- 时间戳决定了在 `put` 请求修改记录时哪些记录更新。
- 时间戳决定了一条记录的多个版本在返回时的排序。
- 时间戳还用于大合并（Major Compaction）过程，决定是否移除与时间戳相比已超过存活时间（Time-To-Live，TTL）的过期记录。“过期”意味着记录的值已经被其他的 put 操作重复写入过，或者该记录已被删除。

默认情况下，写入和更新记录要使用集群节点上那个时刻的时间戳。大多数情况下，这也是很正确的选择，但某些情况例外。比如，实际情况下交易真正发生的时间通常会和 HBase 的记录时间存在几个小时或几天的延迟。这种情况下，通常以交易实际的发生时间设置时间戳。

1.3.3 hop

hop 指在 HBase 中检索请求信息时，所采用的同步 `get` 请求的数量。

让我们来看一个例子。例子中的表格为一个 HBase 表，展现了人与人之间的关系。表 1-2 是一张个人信息表，包含了姓名、朋友名单以及每个人的住址。

表1-2：个人信息表（Person）

姓名	好友	地址
David	Barack，Stephen	唐宁街 10 号
Barack	Michelle	白宫
Stephen	Barack	塞克斯街 24 号

现在，再次回顾拿来做类比的哈希表。如果想查找 David 所有朋友的住址，你需要发起两次 hop 请求。第一个 hop 获取 David 所有朋友的列表，而第二个 hop 查询 David 的朋友的住址。

我们来看另一个例子。表 1-3 为一张学生表，包含学生的学号和名字。表 1-4 为一张课程表，包含学生的学号和每名学生所选的课程。

表1-3：学生表

学号	学生名字
11001	Bob
11002	David
11003	Charles

表1-4：课程表

学号	课程名称
11001	化学、物理
11002	数学
11003	历史

现在，查找 Charles 所选的课程需要两次 hop 请求。第一个 hop 请求从学生表中检索 Charles 的学号，第二个 hop 请求通过学号检索 Charles 所选课程的清单。

正如 1.2.2 节中提到的，反向规范化能够减少请求所需要的 hop 数量。上述例子佐证了这种方法的可取之处。

总之，尽管 HBase 可能需要多个 hop 请求，但最好还是通过更优秀的模式设计避免多次 hop 请求（比如，利用反向规范化）。这是因为每一个 hop 都是从访问 HBase 到返回数据的来回，会有显著的性能消耗。

1.3.4 表和Region

另一个影响性能与数据分布的因素是 HBase 中表的数量以及每个表的 Region 数量。如果分配不合理，集群一个节点或多个节点的负载会出现显著的不均衡。

图 1-3 为含有三个节点的 HBase 集群中，Region 服务器、Region 与表的拓扑结构。

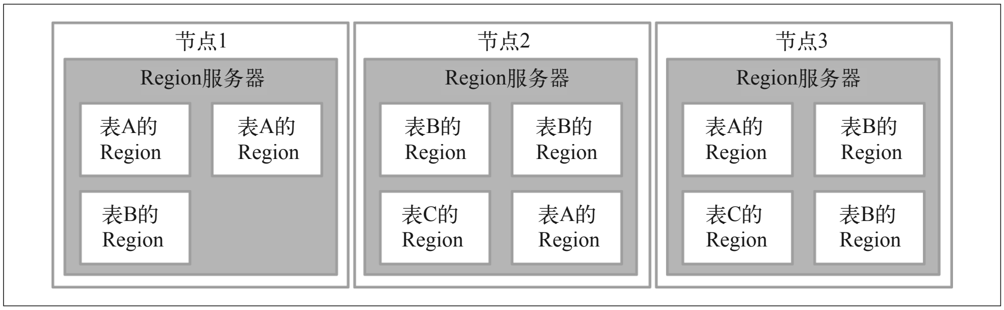


图 1-3：Region 服务器、Region 与表的拓扑结构

以下是值得注意的几点。

- 每个节点包含一个 Region 服务器。
- 每个 Region 服务器包含多个 Region。
- 任何时候，一个给定的 Region 存在一个特定的 Region 服务器上。
- 表被分成多个 Region，而且散布在 Region 服务器中。一个表至少要包含一个 Region。

对于一个给定的表，Region 的数目可以参照两条经验法则进行选择。这两条法则权衡了 `put` 请求的性能与 Region 合并时间。

1. `put`操作的性能

Region 服务器中所有接收到 `put` 请求的 Region 都会共享 Region 服务器的 memstore。memstore 是每个 HBase Region 服务器都有的一种缓存结构。memstore 能缓存发送到 Region 服务器的写入，并对它们进行排序，直到达到特定的内存阈值，冲刷写入磁盘。因此，Region 服务器中的 Region 越多，每个 Region 可用的 memstore 空间就越少。这可能会导致冲刷变小，较小的冲刷又可能带来更小、更多的 HFile 和更多较小的合并，由此导致性能降低。默认的配置将理想的冲刷大小设置为 100 MB，如果确定了 memstore 的大小并分区，使每个区为 100 MB，那么 Region 服务器就会合理地得到最多数量的 Region。

2. 合并时间

Region 越大，合并花费的时间就越多。从经验上来说，Region 的大小至多为 20 GB，但是也有一些非常成功的集群，Region 的大小能够达到 120 GB。

HBase 表分配 Region 的方式有以下两种。

- 默认情况下，一个表只有一个 Region，并随着数据的增加自动分片。
- 创建表时，指定一个 Region 数量，而且将 Region 的大小设置为一个足够大的值（比如每个 Region 100 GB）以避免自动分片。

在选择 Region 分配方式之前，你需要首先确定选择了正确的分片策略。大多数情况下，你应当选择 ConstantSizeRegionSplitPolicy 或者 DisabledRegionSplitPolicy。

大多数情况下，我们建议预先分配 Region 的数量（即采用第二种方式），这样可以避免分片随机进行，避免自动分片导致范围不理想，影响性能。

但是，还有一些情况应该采用自动分片。比如，如果一个不断增长的数据集只更新最新的数据，那么它就更适合采用自动分片。如果该表的行键由 `{Salt}{SeqID}` 组成，那么写操作可能受到控制，分发到一系列固定的 Region 上。既然 Region 自动分片，那么旧的 Region 也就不需要合并了（除非是基于 TTL 的周期性合并）。

1.3.5 使用列

相比传统 RDBMS，列的概念在 HBase 表中大不相同。与 RDBMS 不同，在 HBase 中，一条记录可以拥有上百万个列，而且下一个记录可以拥有百万个完全不同的列。我们一般不建议这么做，但这的确可以实现，而且了解二者的差异也很重要。

为了阐明以上内容，我们先来看一下 HBase 是如何存储记录的。HBase 存储数据的格式被称作 HFile。每个列值在 HFile 中都有自己的行。该行包含行键、时间戳、列名以及列值等字段。HFile 格式还提供了很多其他功能，如版本化和稀疏列存储。但是为了便于说明，下面的例子不涉及这些功能。

比如，如果表含有两个逻辑列，foo 与 bar，那么第一个选择是创建一个 HBase 表，包含两列，名称为 foo 和 bar。这种做法的好处如下。

- 能够一次得到一列，其他列不受影响。
- 能够修改每一列，其他列不受影响。
- 每列都有独立的 TTL。

但是，这些好处也有一定代价。HBase 表中的一条逻辑记录，在 HFile 中会存成两行。磁盘中 HFile 的结构如下。

行键	时间戳	列	值
101	1395531114	F	A1
101	1395531114	B	B1

另一个选择是把 foo 和 bar 的值放在同一个 HBase 列中。这种选择适用于表中的所有记录，而且具有以下特点。

- 同一时刻能够获取两列的值。如果有一列不需要，则可以选择放弃那一列。
- 因为两个列值作为一个单一实体（列）存储，所以更新任何一个列，整个列都会更新。
- 基于最后一次更新时间，两列的 TTL 相同。

这种情况，HFile 的结构如下。

行键	时间戳	列	值
101	1395531114	X	A1|B1

可以占用的磁盘空间大小也决定了如何设计 HBase 的模式，尤其是列的数量。它决定了以

下几点。

- 数据块缓存中能够存放多少记录；
- 多少数据能进入 HBase 维持的 Write-Ahead-Log 中；
- memstore 的一次冲刷可以写入多少记录；
- 合并需要多长时间。

注意，之前例子中列的名称为一个字符。在 HBase 中，就像你所看到的，列与行键的名称在 HFile 中占据了空间。这里建议尽量节省空间，所以通常以单个字符为列名。

1.3.6 列簇

除了列，HBase 中也包含*列簇*（column family）的概念。列簇本质上是列的存储容器。一张表可以有一个或多个列簇。每个列簇都有自己的 HFile 集合，而且在执行合并操作时，同一个表中的其他列簇不受影响。

在很多使用案例中，一张表不需要多个列簇。如果一张表中的一部分列操作完成，或者变化频率与其他列存在显著不同，则可以使用一个以上的列簇。

比如，HBase 表含有两列：列 1 每行包含 400 个字节，而列 2 每行包含 20 个字节。现在我们假设列 1 的值只设置一次，不会改变，但是列 2 的值要经常改变。另外，从访问模式上看，对列 2 调用的 get 请求远多于针对列 1 的 `get` 请求数。

这种情况下，采用两个列簇更好，原因如下。

- *降低合并成本*

  如果有两个独立的列簇，那么包含列 2 的列簇会经常刷新 memstore，所以会产生较小的合并。因为列 2 在其自身的列簇中，所以 HBase 只需要合并总记录中 5% 的数据，因此合并对性能的影响更小。

- *更好地使用数据块缓存*

  就像先前看到的，从 HBase 中检索数据时，附近（位于同一个 HBase 缓存中）的记录会拉入数据块缓存中。如果列 1 与列 2 在同一个列簇中，每次对列 2 调用 `get` 请求时都会把两列数据拉到缓存中。缓存包含了列 1 数据，而列 1 中的数据接受的 `get` 请求非常少，使用的频率也非常低，这就导致了不理想的缓存分布。使列 1 和列 2 位于不同列簇，会导致缓存中填充的数据仅来自于列 2，因此增加了随后对列 2 调用 get 请求的高速缓存命中率。

1.3.7 TTL

TTL 是 HBase 的一个内置特性，即基于数据的时间戳将数据清除。这里的思路是，数据只需存储一定时间，在这段时间里方便使用即可。所以，如果发生大合并时，时间戳早于过去指定的 TTL，那么这些早期的记录不会纳入大合并产生的 HFile 中；也就是说，随着数据的修整，旧的记录被移除了，不再作为表的正常数据。

如果没有使用 TTL，而且仍然有删除历史数据的需求，那么就需要更多的 I/O 密集操作。比如，要移除所有超过 7 年的数据，如果不使用 TTL，那么你就必须扫描 7 年中每天的所有数据，然后对每一条超过 7 年的记录插入一条删除记录。你不仅要扫描 7 年全部的数据，还要创建新的删除记录，这些记录本身的大小就达到了几个 TB。而且，你仍然需要运行大合并，才能最终将那些记录从磁盘中移除。

最后需要说明的是，TTL 基于 HFile 记录的时间戳。如前所述，该时间戳默认为记录添加到 HBase 的时间。

1.4 元数据管理

目前为止，我们已经讨论了 Hadoop 中最好的数据结构化设计方式，以及数据存储方式。然而，关于数据的元数据与数据同等重要。本节将讨论 Hadoop 生态系统中元数据的不同格式，以及如何充分地利用它们。

1.4.1 什么是元数据

一般来说，元数据指关于数据的数据。在 Hadoop 生态系统中，元数据有很多种。下面先罗列几个，如下所述。

- *与逻辑数据集有关的元数据*

  包括以下信息：数据集的位置（比如 HDFS 中的目录或者 HBase 中表的名称）、与数据集有关的模式[1]、数据集的分区与排序特性（如果有的话），以及适用的数据集格式（比如 CSV、TSV、SequenceFile，等等）。此类元数据通常存储于独立的元数据仓库中。

- *与 HDFS 文件有关的元数据*

  包括以下信息：该类文件的权限与属主，以及数据节点上不同数据块的位置。此类信息通常通过 Hadoop NameNode 进行存储和管理。

- *与 HBase 表相关的元数据*

  包括以下信息：表的名称、相关名称空间、相关属性（如 `MAX_FILESIZE`、`READONLY`，等等），以及列簇的名称。此类信息由 HBase 存储和管理。

- *与数据输入和转化有关的元数据*

  包括以下信息：创建指定数据集的特定用户、数据集的来源、创建数据集花费的时间，以及存在多少条记录，或者加载的数据大小是多少。

- *与数据集统计相关的元数据*

  包括以下信息：数据集中行的数量、每列中特定值的数量、数据分布的直方图以及最大值和最小值。此类元数据用于不同的工具，这些工具能够利用元数据优化执行计划。它们也能供数据分析师使用，他们可以基于元数据进行快速分析。

注 1：注意，Hadoop 为 Schema-On-Read。引入模式不代表失去该特性，只是表明这是对数据集中数据的一种解析方式。同一份数据可以拥有多份模式。

本节将讨论之前清单中的第一点：与逻辑数据集有关的元数据。从这里开始，本节所提到的*元数据*，指的均是这一种元数据。

1.4.2　为什么元数据至关重要

有三个重要的理由，让我们不得不在意元数据。

- 元数据允许用户通过一张表的高一级逻辑抽象，而不是 HDFS 中文件的简单集合，或者 HBase 中的表来与数据交互。这意味着用户不必关心如何数据是如何存储的，存储到了什么地方。
- 元数据允许用户提供数据的信息（如分区或者排序特性），而后通过不同的工具（用户或者其他人写入的）利用这些信息生成或者查询数据。
- 元数据允许数据管理工具链接该元数据，而且允许用户执行数据查找（查找可用的数据，并查找如何使用该数据）与数据血缘分析（追踪一个给定数据集的来源或者起源）。

1.4.3　元数据的存储位置

Hadoop 生态系统中第一个开始存储、管理并使用元数据的项目是 Apache Hive 项目。Hive 将元数据存储在一个名为 Hive metastore 的关系型数据库中。注意，Hive 还包涵了名为 Hive metastore service 的服务，它与 Hive 元数据数据库相交互。为了避免弄混数据库和进入该数据库的 Hive 服务，我们将前者称作 Hive 元数据数据库（Hive metastore database），将后者称作 Hive 元数据服务（Hive metastore service）。本书提到 Hive metastore 时，指的是包含服务与数据库的逻辑性集合系统。

随着时间的推进，有意使用 Hive metastore 中元数据的项目越来越多。为了能够在 Hive 以外使用 Hive metastore，有一个名为 HCatalog 的项目启动了。现在，HCatalog 成为了 Hive 的一部分。很重要的一点是，HCatalog 能够使其他工具（比如 Pig 和 MapReduce）与 Hive metastore 整合在一起使用。同时，HCatalog 还通过 WebHCat 服务器将 REST API 加入到 Hive metastore，打开了访问 Hive metastore 的通路，拓展了它的生态系统。

用户可以把 HCatalog 当作访问 Hive metastore 的跳板，如图 1-4 所示。

现在 MapReduce、Pig 与独立的应用程序都能通过相应的 API，很好地与 Hive metastore 取得直接联系。但是 HCatalog 更便于通过 WebHCat Rest API 访问 Hive metastore，而且允许集群管理者出于安全性的考虑对 Hive metastore 的访问加锁。

注意，在使用 HCatalog 和 Hive metastore 时并不一定要用到 Hive。仅在存储和访问与数据相关的元数据时，HCatalog 才会使用 Hive 的基础部分。

Hive metastore 可以在三种模式下使用：嵌入式 metastore、本地 metastore 和远程 metastore。尽管无法详述所有的模式，但是我们建议以远程 metastore 模式使用 Hive metastore。这也是在 Hive metastore 上使用 HCatalog 的一个要求。在常用的数据库中，Hive metastore 可选的数据库有 MySQL、PostgreSQL、Derby 与 Oracle。MySQL 是目前为止行业中应用最为广泛的数据库。当然，可以运行新的数据库实例，为 Hive 创建一个用户并且分配合适的权限，并将该数据库当作自己的 metastore 来使用。如果公司中已经有一个可以使用的关系

型数据库实例，那么可以选择将该数据库实例当作 Hive metastore 的数据库来使用。使用者在已有的数据库实例上为 Hive 创建一个新用户并分配权限即可。

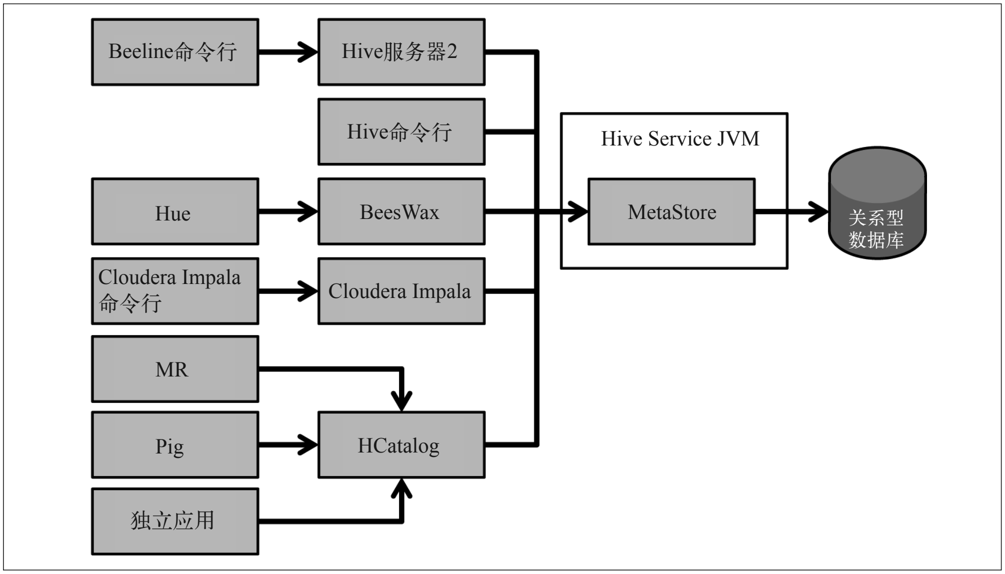


图 1-4：HCatalog 可以作为访问 Hive metastore 的跳板

应该重用现存的数据库实例，还是重新创建一个，取决于数据库实例的使用模式、现存负载，以及使用数据库实例的其他应用。一方面，从操作角度来看，不应该为每个新应用（这种情况下，Hive metastore 处理 Hadoop 生态系统中的元数据）都创建新的数据库实例。但是另一方面，最好不要让 Hadoop 的基础部分交叉依赖于一个十分不匹配的数据库实例。其他考虑因素也很重要。比如，如果公司中已经存在一个可用性极佳的数据库集群，而且你想使用该集群获得 Hive metastore 数据库的高可用性，那么为 Hive metastore 数据库使用现存的 HA 数据库集群更好。

1.4.4　元数据管理举例

使用 Hive 或者 Impala 时，不需要采取任何特殊的操作来创建或者检索元数据。这些系统直接与 Hive metastore 进行整合，这意味着 `CREATE TABLE` 命令就能创建元数据，`ALTER TABLE` 就能改变元数据，表的查询就能够检索存储的元数据。

使用 Pig 时，整合 HCatalog 与 Pig 可以存储和检索元数据。如果使用 Hadoop 中的项目接口（如 MapReduce、Spark 或者 Cascading）来查询数据，可以使用 HCatalog 的 Java API 来存储和检索元数据。HCatalog 同样含有一个 CLI 和一个 REST API，可以用来存储、更新和检索元数据。

1.4.5　Hive metastore与HCatalog的局限性

Hive metastore 和 HCatalog 在应用上存在一些缺点，其中一部分结如下。

- *高可用性相关的问题*

  想为 Hive metastore 提供 HA，就要为 metastore 数据库与 metastore 服务提供 HA。metastore 数据库是每天结束时的数据库，而且已经解决了为数据库提供 HA 的问题。采用任何一种 HA 数据库集群办法都可以为 Hive metastore 数据库提供 HA。只要 metastore 服务持续进行，就能不断支持集群中一个或多个节点上的 metastore 运行。但是，在本文写作的时候，这种做法会带来同步的问题，这个问题关乎数据定义语言（Data Definition Language，DDL）表述以及同一时间运行的其他查询[2]。Hive 社区正在努力解决这些问题。

- *固定的元数据模式*

  Hadoop 中的数据有多种存储类型，对于这一点，Schema-on-Read 概念起到了重要作用。然而，因为关系型后端能够支持 Hive metastore，所以 Hive metastore 中元数据的模式是固定的。现在，因为模式有很多种，所以实情也没有听上去那么糟糕。各种模式都能存储来自列的所有信息，以及区分数据特性的信息类型。有一种比较罕见的情形：你需要存储并检索数据元信息，而元信息又不能存储在 metastore 模式中。这种情况下可能需要为元数据选择别的系统。而且，metastore 旨在为数据集提供一张抽象表格。如果把数据放到一张表中没有什么意义（比如图像或者视频数据），而且仍然需要存储和检索数据，那么 Hive metastore 可能不是合适的工具。

- *另一个存在变数的环节*

  虽然这并不一定意味着限制，但是还是需要记住，Hive metastore 给你的基础设施带来了变数。因此，在使用中你需要保证 metastore 服务的正常运行，而且要像对待其他 Hadoop 基础部分一样，保证 metastore 数据的安全。

1.4.6　其他存储元数据的方式

使用 Hive metastore 与 HCatalog 是 Hadoop 生态系统中存储与管理表元数据最常见的方法。但是，其他情况下（存在先前提到的那些限制），用户更喜欢在不使用 Hive metastore 和 HCatalog 的情况下存储元数据。本节将讨论这些方法。

1. 将元数据嵌入到文件路径和文件名中

在 1.2 节中，你可能已经注意到了，我们建议将一些元数据嵌入数据集位置以保障组织性和一致性。比如，在一个分区数据集中，目录结构如下：

< 数据集名称 >/< 分区列列名 = 分区列值 >/{ 文件名 }。

这种结构已经包含了数据集的名称、分区列的名称，以及数据集分区后分区列的各种值。处理数据时，多种工具与应用都可以使用文件名称与位置中的元数据。比如，想要列出名为 medication_orders 的数据集的所有分区，只要对 HDFS 目录 /data/medication_orders 调用一个 ls 操作即可。

注 2：例子可参见 Hive-4759（https://issues.apache.org/jira/browse/HIVE-4759）。

2. 将元数据存储在HDFS上

你可能会决定将元数据存储于 HDFS 中。存储该类元数据的一种做法是创建隐藏的目录，如 .metadata，它隐藏在包含 HDFS 数据的目录中。你也可能决定将数据模式存储于 Avro 模式文件中。当不能通过 HCatalog 将元数据存储于 Hive metastore 时，这一做法会非常实用。Hive metastore 对于能够存储的数据进行了模式上的规定，HCatalog 也是如此。如果你想要存储的元数据不在范围之内，那么一个合理的选择是管理自己的元数据。比如，HDFS 中的目录结构可能如下：

```
/data/event_log
/data/event_log/file1.avro
/data/event_log/.metadata
```

值得注意的是，如果使用这条路径，那么你将创建、维护和管理自己的元数据。不过，你也可以选择使用类似于 Kite SDK[3] 的工具来存储元数据。而且，Kite 支持多个元数据提供者。这表示在使用该工具在 HDFS 中存储元数据（就像刚刚描述的）的同时，你也可以使用该工具在 HCatalog（以及 Hive metastore）中存储元数据。你可以轻松地将一种来源（HCatalog）的元数据转换为另一种来源（如 HDFS 中的 .metadata 目录）的元数据。

1.5 结论

数据模型的构建在任何一种系统中都是一个很有挑战的任务。Hadoop 生态系统提供了诸多选择，所以挑战尤为明显。之所以会有这么多选择，一个原因是 Hadoop 的灵活性在不断增加。尽管 Hadoop 仍然属于 Schema-on-Read，但选择正确的模型存储数据可以带来很多好处，比如减少存储痕迹、提高处理时间、简化授权与许可、使元数据管理更加便捷。

正如本章所讨论的，你可以自己选择存储方式，其中 HDFS 与 HBase 是最常用的。如果使用 HDFS，那么用来存储数据的文件类型有很多，其中 Avro 是非常常用的面向行型格式，ORC 与 Parquet 是常用的面向列型格式。使用 HDFS 数据时，还应该选择合适的压缩编解码，Snappy 是最常用的选择之一。如果你在 HBase 中存储了数据，从建模角度来看行键的选择可能是最重要的设计因素。

下一个重要的模型选择关乎元数据的管理。元数据可以指很多类型的数据，本章将重点放在了模式相关的元数据，以及数据的字段类型上。Hive metastore 已经成为存储和管理元数据的默认标准，但是你也可以自己管理元数据。

你需要选择正确的数据模型，这是应用中最重要的选择之一。我们希望你能花些时间和精力，争取一次搞定。

注 3：Kite（http://kitesdk.org/）是一个工具集合，包括一系列软件库、工具、示例及文档，关注在 Hadoop 生态系统之上更为便捷地构建系统。它能够让使用者像本节提及的那样，在 HDFS 上创建和管理元数据。

第 2 章

Hadoop数据移动

我们已经围绕 Hadoop 存储与建模讨论了很多需要考虑的因素。接下来，我们将讨论一个同样重要的话题，即外部系统与 Hadoop 之间的数据传递。本章内容包括从外部系统（如关系型数据库或者日志）采集数据导入 Hadoop，以及从 Hadoop 中提取数据导入外部系统。本章将用大量篇幅讲解把数据导入 Hadoop 时所需考虑的因素，以及最佳导入方法。另外，我们还会深入挖掘数据导入的特定工具，如 Flume 与 Sqoop。最后，我们会讨论将数据导出 Hadoop 所需考虑的因素，并提出相关建议。

2.1　数据采集考量

第 1 章谈到了 Hadoop 中数据存储的各种决策。本章将讨论同样重要的架构决策——如何将数据导入 Hadoop 集群。虽然 Hadoop 提供了文件系统客户端，便于在 Hadoop 中和 Hadoop 外复制文件，但是大多数在 Hadoop 上实施的应用都需要从不同来源导入完全不同的数据类型，而且不同的导入频率也提出了不同的要求。Hadoop 常用的数据来源包括以下几个。

- 传统数据管理系统，如关系型数据库与主机。
- 日志、机器生成的数据，以及其他类型的事件数据。
- 从现有的企业数据存储系统中输入的文件。

将数据从不同的系统输入 Hadoop 时需要考虑很多因素。企业导入到 Hadoop 的数据越多，这些决策就越重要。本章将讨论的考虑因素如下。

- 数据采集的时效性与可访问性

  需要采集的数据在采集频率方面有哪些要求？下游的处理要求数据多长时间准备完毕？

- 增量更新

  如何添加新数据？需要将新数据添加到现有数据中吗？需要重写现有数据吗？

- 数据访问和处理

  数据会用于处理过程吗？如果会，数据会用于批处理任务吗？需要的数据是不是随机获取的？

- 数据的来源系统及数据结构

  数据来自哪里？是来自关系型数据库还是来自日志？数据是结构化的、半结构化的还是非结构化的？

- 数据分区及数据分片

  数据采集后应该如何分区？需要将数据导入到多个目标系统（如 HDFS 与 HBase）吗？

- 存储格式

  数据存储的格式是哪一种？

- 数据变换

  需要变换尚未落地的数据吗？

我们首先来看一下第一个考虑因素：数据采集的时效性要求。

2.1.1 数据采集的时效性

提到数据采集的时效性，我们指的是可进行数据采集的时间与 Hadoop 中工具可访问数据的时间之间的间隔。采集架构的时间分类会对存储媒介和采集方法造成很大的影响。为了集中探讨话题，本章不会涉及流处理或者分析，我们会把这部分内容单独放到第 7 章中讲述。本章仅讨论 Hadoop 中数据存储的时间与数据可用于处理的时间。

一般来说，在设计应用的采集构架之前建议使用以下分类中的一种。

- 大型批处理

  通常指 15 分钟到数小时的任务，有时可能指时间跨度达到一天的任务。

- 小型批处理

  通常指大约每 2 分钟发送一次的任务，但是总的来说不会超过 15 分钟。

- 近实时决策支持

  指接受信息后“立即作出反应”，并在 2 秒至 2 分钟内发送数据。

- 近实时事件处理

  指在 2 秒之内处理任务，速度可达到 100 毫秒。

- 实时

  这个词的用途非常广泛，而且有很多定义。“实时”在这里指不超过 100 毫秒。

需要注意的是，随着任务的实现时间达到实时，实现的复杂性和成本也会大大增加。从批量处理出发（比如使用简单文件传输）通常是个不错的选择。选择更加复杂的采集方法之

前要先使用简单的方法。

HDFS 对时效性的要求比较宽松，所以可能更加适合成为主要存储位置。而一个简单文件传输或者 Sqoop 任务则适合作为采集数据的工具。我们将在后面的内容中详细讨论这些选择。这些方法直接提供的实现和验证检查比较简单，所以非常适合批量采集数据。比如，执行 `hadoop fs -put` 命令将复制一个文件，并进行全面的校验，以确定正确地复制了数据。

使用 `hadoop fs -put` 命令与 Sqoop 时，你需要明白一点：HDFS 上的数据存储格式可能并不适合数据的长期存储与处理。因此，在使用这些工具的时候，可能需要通过额外的批处理操作，以将数据存储为需要的格式。比如，将 Gzip 文件加载到 HDFS 中就需要这种额外的批处理操作。尽管 Gzip 文件很容易存储在 HDFS 中，并通过 MapReduce 或其他 Hadoop 处理框架进行处理，但正如我们在前一章中所谈到的，Gzip 文件是不可分片式的。这将对文件处理的有效性造成很大的影响，尤其是当文件变得更大时。这种情况下，一个很好的解决方法是采用容器格式将文件存储至 Hadoop 中。该格式支持可分片式压缩，比如 SequenceFile 或 Avro。

当用户的需要从简单的批处理转向更高频率的更新时，就应该考虑 Flume 或 Kafka 之类的工具了。在这里，传输时间要求不超过 2 分钟，所以 Sqoop 与文件转换器不适用。而且，因为要求时间不超过 2 分钟，所以存储层可能需要变为 HBase 或 Solr，这样插入与读取操作会获得更细的粒度 。当要求提高到实时水平时，我们首先需要考虑内存，然后是永久性存储。全世界所有的平行化处理都不会有助于将反应要求控制在 500 毫秒以内，只要硬盘驱动器保持处理操作的状态。基于这一点，我们开始进入流处理领域，采用 Storm 或 Spark Streaming 之类的工具。这里要强调的是，这些工具应该真正用于数据处理，而不是像 Flume 或 Sqoop 那样用于数据采集。同样，我们也会更多地讨论如何采用类似第 7 章中 Storm 和 Spark Streaming 这样的工具处理接近实时的数据流。本章将深入讨论 Flume 与 Kafka。

2.1.2　增量更新

这里的重点是，新的数据是要添加到已有数据集中，还是要修改已有数据集。如果仅要求添加数据（比如说，将用户活动添加到网络日志），那么 HDFS 对于大部分实现都很适用。HDFS 能够并行化多个驱动器的 I/O 操作，所以读写性能很高。HDFS 的缺点是无法添加或者随机写入创建后的文件。在要求“仅添加”的情况下，这不是什么问题。这种情况下可以通过增加新的文件或者分区来“添加”数据，使得 MapReduce 和其他处理任务能够访问新数据和旧数据。

> 注意，关于分区和 HDFS 中的数据存储，第 1 章进行了详细的讨论。

非常值得注意的是，HDSF 很适合用于将大文件添加到目录中。如果要求添加处理在 2 分钟内完成，而且处理过程产生了很多小文件，那么需要采用一个周期性处理来合并小文件，这样才能保证大文件的优势。选择大文件的理由有很多，其中很重要的一点是从磁盘中读取数据的方式。相比于执行多次查询以从多个文件读取相同信息，采用长期、连续扫描的

方式读取单个文件的速度更快。

> 本章随后会简单讨论一下管理小文件的方法。但是本书不会全面讨论这个问题。想要获知管理小文件技术的详细内容，请参阅《Hadoop 权威指南》或《Hadoop 硬实战》。

如果还需要修改现有数据，那么就要进行额外的工作，将数据存储至 HDFS。HDFS 仅能读取数据，而不能像关系型数据库一样原地更新。因此，我们要首先要写入一个 delta 文件，里面包含要对现有数据做出的修改。接下来要进行一个合并任务（compaction job），处理这些修改。合并任务通过主键对数据分类。如果行被发现了两次，那么就保留来自新 delta 文件的数据，而不保留来自旧文件的数据。合并处理的结果会写入磁盘。当处理完成时，合并后的数据会取代未合并的旧数据。注意，这里说的合并任务是一种批处理操作，比如执行作为任务流一部分的 MapReduce 任务。根据数据的大小，处理可能需要几分钟。因此，这种处理只适合时效性间隔为数分钟的数据。

表 2-1 为一个执行合并任务的例子。注意，第一列为数据的主键。

表2-1：合并

原始数据	新增数据	结果数据
A, Blue, 5	B, Yellow, 3	A, Blue, 5
B, Red, 6	D, Gray, 0	B, Yellow, 3
C, Green, 2		C, Green, 2
		D, Gray, 0

采用不同的方法执行合并操作时，性能会不同。第 4 章将讲解执行合并操作的例子。

另外一个选择是不考虑 HDFS 与文件合并，转而使用 HBase。这种情况下 HBase 能够执行合并操作，并且在架构上支持增量。HBase 中 `put` 命令完成时，该构架即可生效，而且通常在毫秒之内发生。

需要注意的是，使用 HBase 需要其他的管理与应用开发知识。与 HDFS 相比，HBase 的访问模式大不相同，比如扫描速度。HBase 的扫描速度大约是 HDFS 的 1/10~1/8。另一个不同点是随机访问。HBase 访问单个记录的时间为毫秒级别，而 HDFS 不支持随机访问，只支持昂贵、复杂的文件查询。

2.1.3　访问模式

这里要求深入理解信息传输的潜在要求。一旦存储至 Hadoop 中，数据将如何使用？比如，如果需要获得随机行访问，HDFS 可能不是最好的选择，这种情况更适合选用 HBase。相反，如果要求扫描与数据转化，HBase 就不那么合适了。这里需要考虑的因素很多，我们提出一个基本的指导原则：如果要求操作简单、压缩性能最优、扫描速度最快，则默认选择 HDFS。另外，新版本的 HDFS（Hadoop 2.3.0 以及更高版本）支持将数据缓存至内存中。所以，工具能够直接从内存中读取数据，然后将文件加载到缓存中。该作用使 Hadoop 成为了支持大批量并行处理的内存数据库，能够被 Hadoop 生态系统中的工具访问。当主

要需求是随机访问时，应该默认选择 HBase，有检索处理的需求时，可以考虑选择 Solr。

更多内容详见表 2-2，其中包括这些选择的常见访问模式。

表2-2：Hadoop存储类型的访问方式

工具	用途	存储设备
MapReduce	大型批处理	推荐使用 HDFS，也可使用 HBase（非首选）
Hive	类 SQL 接口的批处理	推荐使用 HDFS，也可使用 HBase（非首选）
Pig	采用数据流语言的批处理	推荐使用 HDFS，也可使用 HBase（非首选）
Spark	快速交互式处理	推荐使用 HDFS，也可使用 HBase（非首选）
Giraph	图的批处理	推荐使用 HDFS，也可使用 HBase（非首选）
Impala	MPP 架构上的 SQL 引擎	大多数情况下应选择 HDFS。但是在某些情况下，即使扫描速度降低，也应当采用 HBase。比如，要对更新的数据近实时访问，通过主键快速访问记录
HBase API	对记录级别的数据原子性地执行 `put`、`get` 与 `delete` 操作	HBase

2.1.4 数据源系统及数据结构

从文件系统中采集数据时，应该考虑以下内容。

1. 数据源系统设备的读取速率

在所有处理流水线中，磁盘 I/O 通常都是主要瓶颈。这一点可能并不明显，但是优化采集流程时通常要看一下检索数据的系统。一般来说，Hadoop 的读取速度在 20 MB/s 到 100 MB/s 之间，而且主板或者控制器从系统所有的磁盘中读取数据时有一定的限制。为了使读取速度达到最高，需要确保尽量充分利用系统中的磁盘。某些网络附加存储（Network Attached Storage，NAS）系统会通过额外增加挂载点来加大吞吐量。同样要注意的是，一个单一的读取线程不会提升驱动器或设备的读取速度。根据我们的经验，一个典型的驱动器通常要使用三线程才能使吞吐量达到最大。但是这个数字会随具体情况的不同而有所改变。

2. 原始文件格式

数据可以为任何一种格式：带分隔符文本、XML、JSON、Avro、定长文件、变长文件、Copybook，等等。Hadoop 能够接受任意一种文件格式，但是并不是所有格式都适合特定的使用案例。举个例子，想想 CSV 文件。这是一种非常常见的格式，而且这种格式的文件通常很容易导入一张 Hive 表，进而可以立刻访问和处理数据。但是，很多进行 CSV 文件底层存储格式转换的任务能够（通过格式转换）提供更优化的数据处理。比如，使用 Parquet 作为存储格式进行数据分析可以提供更有效的处理，同时也能减小文件的存储空间。

另外需要考虑的是，Hadoop 生态系统中的这些工具并不能支持所有的文件格式，比如变长文件。某些平面文件（flat file）的列数是固定的。变长文件与之类似。定长文件与变长文件的差异在于，后者最左侧的一列决定后续文件读取的规则。比如，最开始的两列是 8 字节的 ID，随后是一个 3 字节的类型字段。ID 只是一个全局标识符，读取数据的方式与定长文件相似。但是，类型字段设定了该记录其余内容的读取规则。如果类型字段的值为

`car`，那么记录可能包含最大速度、里程、颜色之类的列；如果值为 `pet`，那么记录中的列可能为大小、品种，等等。不同的列长度不同，因此称作“可变长度”。理解了以上内容，我们就能明白变长文件可能不适合 Hive，但是仍然能以 Hadoop 中的处理框架进行有效处理，这样的处理框架包括 MapReduce 的 Java 任务、Crunch、Pig 和 Spark。

3. 压缩格式

在原始文件系统对数据进行压缩的做法有优点也有缺点。其优点在于，通过网络传输压缩文件较为节省 I/O 和网络宽带。其缺点在于大多数适用于 Hadoop 之外的压缩编码器都不支持分片（如 Gzip）。不过，在 Hadoop 中使用可分片的容器格式，可以使这些编码支持分片。正如第 1 章提到的，这些容器格式包括 SequenceFile、ParquetFile 和 AvroFile。通常这种做法首先会将压缩文件复制到 Hadoop 中，后续处理时再对文件进行转换操作。转换操作也可以在数据输入 Hadoop 时进行，但使用 Hadoop 集群的分布式处理能力转换文件，比仅仅使用数据传输到集群中时涉及的边缘节点更好。

> 第 1 章讨论了 HDFS 中数据压缩需要考虑的因素。本书后面的内容将通过案例，看一下数据采集与后续处理的具体实例。

4. 关系型数据库管理系统

Hadoop 应用通常会整合来自不同 RDBMS 厂商（如 Oracle、Netezza、Greenplum、Vertica、Teradata、Microsoft 等）的数据。这里经常选择的工具是 Apache Sqoop。Sqoop 功能丰富，支持许多选项。相比 Hadoop 生态系统中其他项目，Sqoop 使用起来更为简单便捷。这些选项能控制从 RDBMS 中检索哪些数据，怎样检索数据，使用哪一个连接器，使用多少个 Map 任务，采用怎样的分片模式，以及最终的文件格式。

Sqoop 进行的是批处理操作。如果批处理操作无法满足数据加载到集群中的时效性要求，那么就需要采用其他办法。在某些情况下，你可以选择分片采集数据，以一条流水线将数据加载到 RDBMS 中，以另一条流水线将数据加载到 HDFS 中。Flume、Kafka 之类的工具都能进行这种操作，但是实施起来很复杂，要求在应用层编码。

注意，在 Sqoop 架构下，要与 RDBMS 连接的不只有边缘节点，还有数据节点（DataNode）。有些网络配置不允许出现这种情况，也不可能选择 Sqoop，比如，设备与防火墙之间存在网络瓶颈问题。这些情况下，Sqoop 的替代方案则是生成 RDBMS 的转储文件，然后将文件输入 HDFS 中。大多数关系型数据库都能创建一个带有分隔符的转储文件，而后通过一个简单的文件传输将转储文件输入 Hadoop 中。图 2-1 为 Sqoop 与 RDBMS 转储文件的差异。

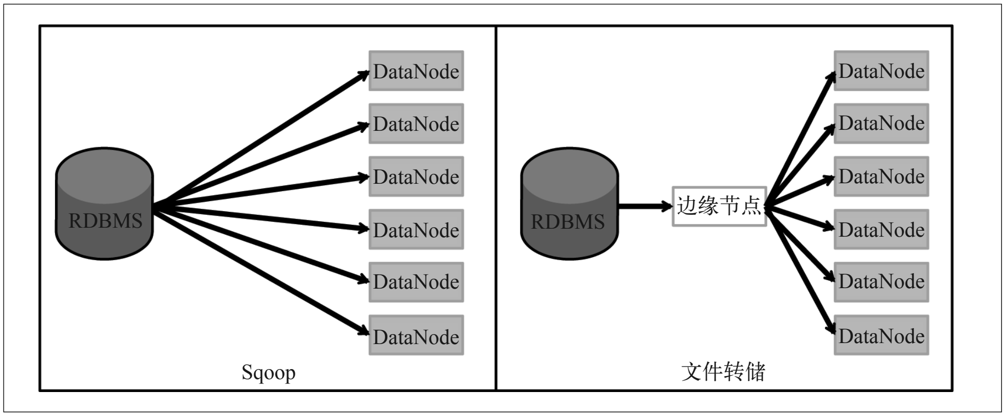


图 2-1：Sqoop 与 RDBMS 转储文件的比较

本章后面将详细介绍更多考虑因素，并推荐使用 Sqoop 的最佳方法。

5. 流式数据

流输入数据包括 Twitter 订阅、Java 消息服务（Java Message Service，JMS）队列，以及网络应用服务器发送的事件。在这种情况下，我们强烈推荐使用 Flume 或 Kafka。这两个系统都能提供同样水平的保证，而且功能相似，但它们之间存在一些重要的差异。随后我们会在本章深入了解 Flume 与 Kafka。

6. 日志文件

文件系统与流输入之间的部分为日志。我们将单独讨论日志的相关内容。反模式指写入日志时从磁盘中读取日志，因为完成实施却不丢失数据是不可能的。采集日志的正确方法是直接将日志输入工具中，如 Flume 或 Kafka，而不是直接输入 Hadoop。

2.1.5 变换

一般来说，*变换*（transformation）指修改输入数据，将数据分区或者分桶，或者将数据发送到多个存储设备或位置。下面是一些简单的例子。

- *变换*

  将 XML 或 JSON 转换为带分隔符的数据。

- *分区*

  如果输入的是股票交易数据，那么要求按照股票分区。

- *分片*

  将数据加载到 HDFS 与 HBase 中，以获得不同的访问模式。

转换数据的方式要根据时效性的要求进行选择。如果数据的时效性为批处理操作，那么更好的方法可能是先进行文件传输，再进行批转换。注意，这类转换可以通过 Hive、Pig 或 Java MapReduce 之类的工具，或者 Spark 之类的新型工具完成。批转换步骤最好使用后处理，因为该处理过程提供了检测故障的检查点机制，包括文件传输的校验和，以及

MapReduce 的 all-or-nothing（成功 / 失败）行为。有一点一定要注意：能够配置 MapReduce 进行数据的传输、分区与分片，而不需要通过 all-or-nothing 处理。你可以配置带有两个输出目录的 MapReduce，一个目录用于处理成功的记录，另一个处理失败的记录。某些情况下，出于数据的时效性要求，不能进行简单的文件传输，并在其后进行 MapReduce 处理。有时，在数据由 Flume 之类的工具通过流采集输入集群时，需要完成这个任务。如果使用 Flume，这个任务可以通过拦截器（interceptor）和选择器（selector）完成。本章后面将介绍更多和 Flume 相关的知识。这里，我们先简单了解一下拦截器和选择器的作用。

1. 拦截器

Flume 拦截器属于 Java 类，使得事件数据在传输过程中拥有丰富的选择。因为拦截器是通过 Java 语言实现的，所以在能够实现的功能上，它具有极大的灵活性。拦截器能够转换一个事件或者一批事件，也能够从一批事件中增加或移除事件。实施拦截器时必须谨慎，因为它属于流框架的一部分，所以实施不应该引起流水线变慢，变慢的原因包括外部服务器调用和垃圾收集问题。同样需要记住的是，在使用拦截器转换数据时，处理能力有一定的限制，因为 Flume 通常只在集群中的一部分节点上安装。

2. 选择器

在 Flume 中，选择器是一个岔路口。它会决定事件将走哪一条路。我们将在随后的 Flume 模式中提供几个例子。

2.1.6　网络瓶颈

将数据输入 Hadoop 中时，瓶颈常常是源系统或者源系统与 Hadoop 之间的网络。如果瓶颈是网络，那么增加网络带宽和压缩数据就很重要了。你可以在发送文件之前进行压缩处理，或者使用 Flume 压缩网络瓶颈两端的 agent 之间的数据。

如果想要确定网络是否为瓶颈，最简单的方法是看一下吞吐量的数值以及网络配置。1 Gb 以太网的预计吞吐量大约为 100 MBps。如果通过 Flume 看到吞吐量也是如此，那么网络带宽实际上已经得到了最充分的应用。这时就需要考虑压缩数据了。想要得到更精准的判断，你可以使用网络监测工具来测定网络使用量。

2.1.7　网络安全性

用户有时只能从公司防火墙外获取数据。通过有线传输时，有些数据可能需要加密。你可以在发送数据前简单加密文件，如使用 OpenSSL。你也可以使用 Flume，在 Flume agent 之间加密要发送的数据。注意，Kafka 目前并不支持 Kafka 数据流之间的数据加密，所以你需要在 Kafka 之外进行加密和解密工作。

2.1.8　被动推送与主动请求

本章讨论的所有工具可以分为两类：被动推送和主动请求。架构中的 Actor 值得注意，因为最终 Actor 有一些额外的要求需要考虑。

- 已发送数据的跟踪。

- 任务失败后选择重试或故障切换。
- 对数据源系统持敬畏之心，切莫滥用。
- 访问控制与安全性。

本章还会更详细地介绍这些要求，但是我要首先讨论一下两个常用的 Hadoop 工具——Sqoop 与 Flume，以明确使用 Hadoop 采集和提取数据时，数据推送和抽取的差异。

1. Sqoop

Sqoop 是一种数据抽取方法，用于从外部存储系统（如关系型数据库）中提取数据，并输入 Hadoop 中，或者从 Hadoop 中提取数据，并输入外部系统。这里必须提供源系统与目标系统的不同参数，如源数据库的连接信息，需要提取的一张或多张表，等等。参数确定之后，Sqoop 将执行一系列任务，移动所请求的数据。

在以下例子中，Sqoop 从关系型数据库中提取（拉）数据。我们需要考虑很多因素。举例来说，Hadoop 容易消耗源系统中的大量资源，所以要确保按照确定的速度从源数据库中提取数据。我们也需要对 Sqoop 任务做出计划，防止它干扰源系统的加载高峰期。本章后面会提供使用 Sqoop 时的考虑因素与细节，并提供特定的使用案例。

2. Flume

注意，根据使用的源系统，两个版本中的 Flume 都能够被分桶。通常会采用 Log4J appender，这种情况下，Flume 通过流水线推事件。Flume 源也有很多，包括 Spooling Directory 源和 JMS 源，这里事件处于被抽取的状态。再说一下，我们会在本节以及本书后面的案例中更详细地讲解 Flume。

2.1.9　错误处理

设计数据采集流水线时，一个主要的考虑因素是错误处理。出现错误时如何恢复数据是一个非常重要的问题。在大型分布式系统中，问题不在于错误是否会发生，而是什么时候发生。设计采集流时，我们可以设置多层保护（有时会牺牲不少性能），但是没有什么高招能阻止所有的错误。这里需要记录错误场景，包括错误推迟预期，以及如何处理数据丢失。

最简单的错误场景与文件传输有关，比如，`hadoop fs -put <filename>` 命令的执行。`put` 命令执行完毕时，命令完成了文件在 HDFS 中的验证，复制三次，并检查校验和。但是如果 `put` 命令失败了呢？处理文件传输错误的一般方法是，采用多个本地文件系统目录，在文件传输过程中代表不同的分桶。我们来看一下使用该方法的案例。

本例中共有四个本地文件系统目录：ToLoad、InProgress、Failure 和 Successful。这里的工作流很简单，如下所示。

(1) 将文件移动到 ToLoad。
(2) 准备好调用 `put` 命令时，将文件移动到 InProgress 中。
(3) 调用 `put` 命令。
(4) 如果 `put` 命令出现错误，则将文件移动到 Failure 目录中。
(5) 如果 `put` 命令成功，将文件移动到 Successful 目录中。

注意，如果在 `put` 命令执行时出现了错误，那么需要重新发送文件。对大型文件调用 `hadoop fs -put` 命令时，需要谨慎考虑该问题。如果文件传输过程中网络错误发生时间为五个小时，那么需要重新开始整个处理过程。

文件传输非常简单，所以我们来看一个流采集与 Flume 的例子。在 Flume 中，很多区域都可能发生错误，我们把问题简化一下，将重点放在以下三个方面。

- 如果 Channel 已满，数据抽取的源（如 Spooling Directory 源）将无法把事件加载到 Channel 中。这种情况下，在重新尝试进入 Channel 或保留事件之前，该数据源必须停止。
- 如果 Channel 已满，主动接收的数据源将无法将事件加载到 Channel 中。这种情况下，该数据源需要告知 Flume 用户无法接收最后批次的事件，而后 Flume 用户能够自由地再次发送事件至该数据源或其他数据源。
- Sink 无法将事件加载到最后位置。Sink 将从 Channel 中提取大量事件，而后尝试将事件发送到最终位置。但是如果发送失败，那么需要将该批次的事件重新返回到 Channel 中。

Flume 与文件传输之间最大的区别在于，出现错误时，使用 Flume 能够复制 Hadoop 中产生的记录。这是因为，出现错误时，事件总是会返回到最后一步。因此，如果批处理部分成功，我们就将得到复制记录。解决该问题的方法很多，包括使用 Flume 和其他流处理工具，如 Kafka。但是，流处理时尝试将复制记录移除，会带来很大的性能成本。我们会在处理删除的内容中看一个例子。

2.1.10 复杂度

最后一个需要考虑的设计因素是用户使用的复杂性。简单来说，这里需要考虑用户是否需要手动将数据移动到 Hadoop 中。对于这种案例，可以使用 `hadoop fs -put` 命令来解决问题，也可以采用可挂载的 HDFS，如用户空间的文件系统（Filesystem in Userspace，FUSE）或新的网络文件系统（Network File System，NFS）。我们随后会讨论这些选择。

2.2 数据采集选择

我们已经了解了设计数据采集流水线时需要考虑的因素，下面将深入探讨特定的采集工具，以及移动数据到 Hadoop 中的方法。我们会从简单加载文件至 HDFS，一直讲到使用强大的工具，如 Flume 与 Sqoop。如前所述，我们的目的不是深入地介绍这些工具，而是介绍如何有效地使用这些工具，并且将这些工具作为应用架构的一部分。文中提及的各工具的参考资料会提供更为深入和全面的信息。

2.2.1 节会介绍基本的文件传输，而后讨论融入 Hadoop 架构的 Flume、Sqoop 与 Kafka。

2.2.1 文件传输

将数据导入（以及导出）Hadoop 最简单的方法就是采用文件传输——换句话说，就是使用 `hadoop fs -put` 与 `hadoop fs -get` 命令。这有时也是最快的方法。所以，在设计 Hadoop 新的数据处理流水线时，首先应该考虑选择文件传输。

在详细讲解之前，我们先总结一下 Hadoop 文件传输的特点。

- 这是一种 all-or-nothing 批处理方法，所以如果文件传输过程中出现错误，则不会写入或读取任何数据。这种方法与 Flume、Kafka 之类的采集方法不同，后者提供一定程度的错误处理功能，并且有传输保障。
- 文件传输默认为单线程，不能并行文件传输。
- 文件传输将文件从传统的文件系统导入 HDFS。
- 不支持数据转换，数据按原样导入 HDFS。数据导入 HDFS 后才能进行处理，这一点与传输过程中的数据转换截然相反。类似于 Flume 的系统支持传输过程中的数据转换。
- 这种加载是逐字节进行的，所以能传输任何类型 51 的文件（文本、二进制文件与图片等）。

1. HDFS客户端命令

你应该已经注意到了，文件通过 `hadoop fs -put` 命令从一个文件系统复制到 HDFS 中，而且是逐字节地复制。完成 `put` 命令时，文件将作为一个或多个数据块存储在 HDFS 中，每个数据块都通过不同的 Hadoop 数据节点进行复制。复制文件的个数是可配置的，但通常默认复制三次。为了保证数据块不被损坏，每个数据块都会配备一个校验和文件。

使用 `put` 命令时通常可以采用两种方法：*双跳式*（double-hop）与*单跳式*（single-hop）。双跳式如图 2-2 所示，需要从 Hadoop 边缘节点的磁盘多进行一些数据读写，所以这种方法比较慢。尽管如此，如果数据源的外部文件系统无法挂载到 Hadoop 集群中，那么这就是唯一可行的方法。

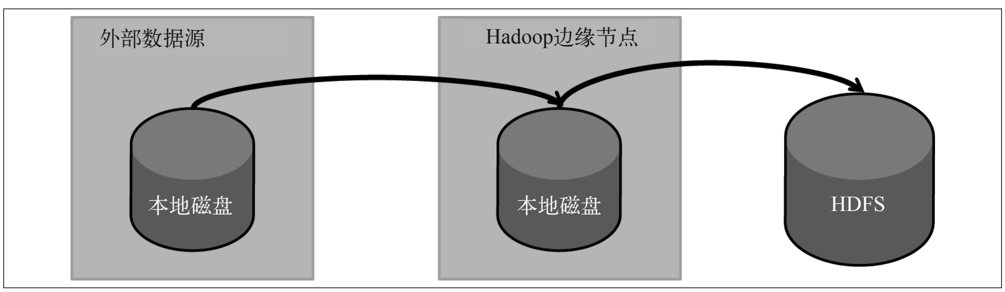


图 2-2：双跳式文件传输

另一种方法是单跳式方法，如图 2-3 所示。单跳式方法要求源数据设备可挂载，比如 NAS 和 SAN。外部数据源可以挂载，而 `put` 命令能直接从设备读取文件，并直接将文件写入 HDFS。这种方法的优点是能够提升性能，也降低了边缘节点对大型本地存储的需求。

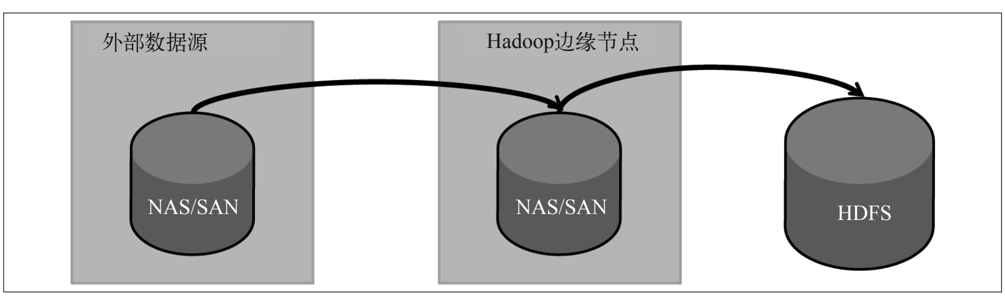


图 2-3：单跳式文件传输

2. 可挂载的HDFS

除了使用 Hadoop 客户端中包含的命令，还有很多选择可以将 HDFS 挂载为一个标准的文件系统。这样一来，用户就可以通过标准文件系统中的命令（包括 `ls`、`cp` 和 `mv`，等等）同 HDFS 交互了。这可以方便用户访问 HDFS，而这里也存在 HDFS 通过客户端使用时会受到的限制。

- 与直接访问 HDFS 相同，这些方法不能提供完整的 POSIX 语义 [1]。
- 这里不支持随机写入；后端文件系统仍然是“一次写入，多次读取”。

可挂载 HDFS 的另一个缺陷是存在误用风险。尽管用户能够更轻松地访问 HDFS 与采集文件，但这样的访问可能会引发很多小文件的采集。因为 Hadoop 在优化之后更适合存储数量相对较少的大文件，所以应该防止这种情况发生。注意，“很多小文件”可能表示，在一个大小合理的集群中存在数百万个文件。而且，为了保证存储性能与处理性能，Hadoop 最好存储数量更少的大文件。如果需要将很多小文件输入 Hadoop，可以采用以下方法降低影响。

- 使用 Solr 存储和检索小文件。第 7 章将深入讨论 Solr。
- 使用 HBase 存储小文件，使用路径和文件名称作为键。第 7 章将深入讨论 HBase。
- 使用容器格式，如 SequenceFiles 或 Avro，合并小文件。

很多项目都为 Hadoop 提供了可挂载的接口，我们重点看一下两个常用的选择：Fuse-DFS 与 NFS。

- Fuse-DFS

  Fuse-DFS 基于 FUSE 项目创建，旨在促进 UNIX/Linux 文件系统的创造，同时不需要修改核心系统。Fuse-DFS 使用户更容易在本地文件系统安装 HDFS。但是作为用户空间模块，Fuse-DFS 包括用户应用与 HDFS 之间的一系列跳跃，会显著地影响性能。另外，Fuse-DFS 的模型持续性较差。因此，用户在使用 Fuse-DFS 解决产品问题之前应该慎重考虑。

- NFSv3

  近来出现的项目增加了 HDFS 支持 NFSv3 协议（如图 2-4 所示）。该设计提供了可扩展的方案，而且性能影响最小。这种设计包括一个 NFS 网关服务器，该服务器能使用 DFSClient 将文件输入 HDFS。用户可以通过增加多个 NFS 网关节点扩展这一方法。

  注意，通过 NFS 网关，核心系统能随机发送写入操作。这就要求 NFS 服务器在将写入操作发送到 HDFS 之前，缓存并记录写入操作。这就导致了较高的数据容量，并且会影响性能和磁盘消耗。在部署任何可挂载的 HDFS 方案之前，一定要先检验一下数据容量是否足够大，以保证该方案能提供所需的性能。总之，仅推荐将这些方法用于少量的手动数据传输，而不能用于数据在 Hadoop 集群进出的移动过程。

注 1：POSIX 是一系列标准，意在使操作系统具有可移植性。其中也包括对文件系统的要求。HDFS 是一种“类 POSIX”的文件系统，但它并不具有 POSIX 兼容文件系统所支持的全部特性。

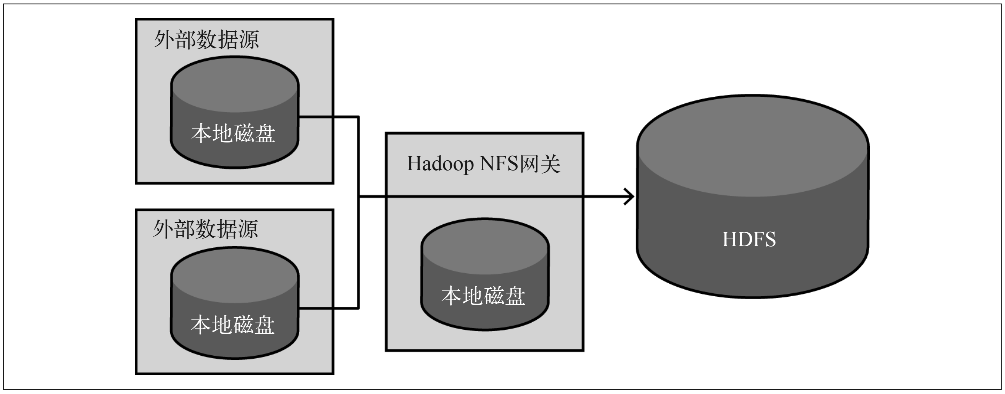


图 2-4：HDFS 的 NFSv3 网关

2.2.2 文件传输与其他采集方法的考量

简单文件传输在某些情况下是适用的，尤其是在需要将已存在的一系列文件输入到 HDFS 中，而且可以接受保持源文件的原格式的情况下。否则，在决定是否可以接受文件传输，或者是否使用类似于 Flume 的工具时，需要考虑以下因素。

- 需要将数据采集到多个位置吗？比如，是需要将数据同时输入 HDFS 和 Solr，还是需要将数据同时输入 HDFS 和 HBase ？这种情况下，如果使用文件传输，那么在文件采集完成之后将需要额外的工作，因此采用 Flume 更为适合。
- 对可靠性的要求高不高？如果高，那么一旦传输过程出现错误，文件传输就必须重新开始。这时，Flume 同样是更好的选择。
- 数据采集之前需要转换操作吗？如果需要，Flume 无疑是适合的工具。

如果需要采集文件，可以考虑的使用 Flume Spooling Directory 源（https://flume.apache.org/FlumeUserGuide.html#spooling-directory-source）。采用这种方法，用户将文件放置到磁盘中特定的目录就可以采集文件。这种采集文件的方法简单可靠，而且需要时能够实现传输过程中的数据转换。

2.2.3 Sqoop：Hadoop与关系数据库的批量传输

本章已经介绍了很多 Sqoop 的知识，下面来详细了解一下相关的考虑因素和最佳实践。如前所述，Sqoop 是一种工具，能批量地将数据从关系数据库管理系统导出到 Hadoop 中，也能批量地将数据从 Hadoop 导出至关系数据库管理系统。当使用 Sqoop 将数据导出至 Hadoop 中时，Sqoop 生成仅限于映射之内的 MapReduce 任务。每个 `Mapper` 使用一个 Java 数据库连接（Java Database Connectivity，JDBC）驱动器连接数据库，然后选择需要导出的表，并将数据写入 HDFS。Sqoop 非常灵活，不仅能够输出全部的表，还增加了 `where` 语句来过滤导出的数据，甚至能提供查询操作。

比如，按照以下方法可以输出单一的表。

```
sqoop import --connect jdbc:oracle:thin:@localhost:1521/oracle \
--username scott --password tiger \
--table HR.employees --target-dir /etl/HR/landing/employee \
--input-fields-terminated-by "\t" \
--compression-codec org.apache.Hadoop.io.compress.SnappyCodec --compress
```

下面为输出一个合并结果的方法：

```
sqoop import \
  --connect jdbc:oracle:thin:@localhost:1521/oracle \
--username scott --password tiger \
--query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id) WHERE $CONDITIONS' \
--split-by a.id --target-dir /user/foo/joinresults
```

注意，在这个例子中，`$CONDITIONS` 是一个文本值，作为命令行的一部分输入。Sqoop 通常使用 `$CONDITIONS` 占位符控制任务的并行化处理，运行命令时 Sqoop 会用产生的条件代替该占位符。

本节将概述几个以 Sqoop 为数据采集方法的模式。

1. 选择可分片的列

输出数据时，Sqoop 会采用多个 `Mapper` 来并行化数据采集过程，增加吞吐量。在默认情况下，Sqoop 使用 4 个 `Mapper`，而且分片处理任务。分片的方法为得到主键列的最大值与最小值，然后在 `Mapper` 之间平均分配值的范围。将 `Mapper` 之间的表平等地分片时，`split-by` 参数能指定不同的列，而且 `nummappers` 参数能控制 `Mapper` 的数量。下面马上会讨论到，在这里指定一个 `split-by` 参数，原因之一是要避免数据倾斜。注意，每一个 `Mapper` 本身都会跟数据库连接，而且在 `where` 语句中指定限定表的一部分时，每一个 `Mapper` 都会检索该表的限定部分。选择一个包含索引或者分区键的分片列很重要，这样能够避免每个 `Mapper` 都扫描整个表。如果没有这类的键，最好选择只指定一个 `Mapper`。

2. 尽可能地使用数据库专用的连接器

不同的 RDBMS 支持不同版本的 SQL 语言。除此之外，不同 RDBMS 会以不同的方法实施和优化数据传输。Sqoop 含有一个通用的 JDBC 连接器，所以能用于任何一种支持 JDBC 的数据库。但是也存在一些供应商指定的连接器，能翻译不同的语言并优化数据传输。比如，Teradata 连接器使用 Teradata 的 FastExport 来以最佳方式执行数据导出，而 Oracle 连接器禁用了并行化查询，以避免查询协调器出现瓶颈。

3. 使用Goldilocks原则调优Sqoop性能

大多数情况下，Hadoop 集群的容量都远胜于 RDBMS。如果 Sqoop 使用了太多 `Mapper`，Hadoop 将针对用户的数据库有效运行一个拒绝服务攻击。如果使用了过多的 `Mapper`，那么数据采集速度可能需要非常慢才能满足要求。这里的要点是调整 Sqoop 任务，以使用合适数量的 `Mapper`——也就是坚持 Goldilocks 原则[2]。

由于数据库超载的风险远大于采集速度变慢的风险，所以我们通常首先采用数量较少的 `Mapper`，然后逐渐增加 `Mapper` 的数量，以达到 Sqoop 采集速度与数据库和网络对所有用户的响应之间的平衡。

注 2：Goldilocks 原则指事物必须落在限定范围之内，而不会达到极限值。

4. 使用公平调度器调整并行加载多张表

从同一个 RDBMS 中采集多张表的情形很常见。有两种使用公平调度器进行节流控制的方法[3]。

- *顺序加载表*

  这是目前为止最简单的方法，其缺点是无法优化 RDBMS 与 Hadoop 集群之间的带宽。为了阐明该方法，在此设想我们拥有五张表。因为并不是所有的表都需要通过全部数量的 `Mapper` 来采集，所以采集每张表所使用的 `Mapper` 数量不同。本例的结果如图 2-5 所示。

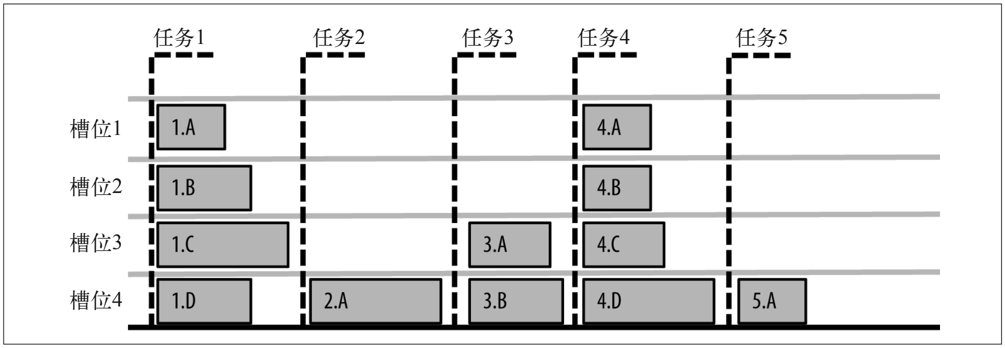


图 2-5：Sqoop 任务顺序执行

  如图 2-5 所示，执行特定任务时，有些 `Mapper` 闲置了。这种情况导致用户产生并行化运行任务的想法，由此便能充分利用可用的 `Mapper`，缩短处理时间与网络时间。

- *并行加载表*

  Sqoop 任务的并行化运行可以使用户更为有效地使用资源，但是管理针对 RDBMS 运行的所有 `Mapper` 的复杂度也有所增加。要解决这个问题，可以使用公平调度器创建一个资源池，将 `Mapper` 的最大数量设定为 Sqoop 与 RDBMS 交互时能够使用的最大数量。如果操作正确，执行同步方法中所提到的五个任务将如图 2-6 所示。

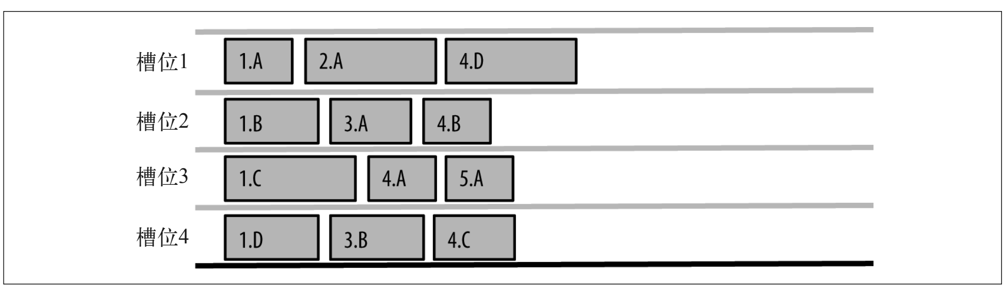


图 2-6：通过公平调度器将任务限制为 4 个，Sqoop 任务并行执行

注 3：公平调度器是一种在任务之间共享资源的方法，能够为每个任务平均分配资源。想要进一步了解更多 Hadoop 公平调度器相关内容及使用方法，请参阅 Eric Sammer 编著的《Hadoop 技术详解》第 7 章。

5. 瓶颈诊断定位

有时我们会发现，数据采集速度并没有随着 Mapper 数量的增加而提升。理想的情况下，Mapper 增加 50%，采集速度也应当每分钟增加 50%。尽管事实上很难达到这个水平，但如果增加 Mapper 对采集速度造成的影响很小，那么流水线中可能存在瓶颈。

以下为一些可能存在的瓶颈。

- 网络瓶颈

  Hadoop 集群与 RDBMS 之间的网络可能为 1 GbE 或 10 GbE。这意味着二者的采集速度分别限制在 120 MBps 和 1.2 GBps 左右。如果接近这个速度，那么增加 Mapper 将会增加数据库的负载，却不能提高采集速度。

- 关系型数据库

  从数据库中读取数据占用了数据库服务器的 CPU 资源，以及磁盘的 I/O 资源。如果无法获得这些资源，那么增加 Mapper 数量可能会使情况变得更差。你需要检查 Mapper 产生的查询。在增量模型中，Sqoop 更适合使用索引。使用 Sqoop 采集整个表时，通常要全表扫描。如果多个 Mapper 竞争访问同一个数据块，那么吞吐量也会降低。最好与企业的数据库管理者探讨一下 Sqoop 的问题，并且让管理者在数据导出到 Hadoop 时监控数据库。当然，在数据库使用量很低时，计划 Sqoop 执行时间是最理想的。

- 数据倾斜

  使用多个 Mapper 输出整个表时，Sqoop 用主键将 Mapper 之间的表分开。该方法先得到键的最高值与最低值，然后按照 Mapper 数量平均分配。比如，如果我们使用了两个 Mapper，customer_id 是表的主键，运行 `select min(customer_id)` 和 `max(customer_id) from customers` 命令得到的值为 1 与 500。那么第一个 Mapper 会输出 1~250 的消费者数据，而第二个 Mapper 输出 251~500 的消费者数据。这意味着如果数据出现倾斜，由于丢失值，部分范围真实含有的数据更少，那么一些 Mapper 就需要处理更多的任务。使用 `--split-by` 命令可以选择更好的分片列，或者使用 `--boundary-query` 命令进行查询，以确定在 Mapper 中将行分片的方法。

- 连接器

  不使用 RDBMS 特定的连接器（connector），或者使用旧版本的连接器，都会导致采用更慢的导入方法，因此采集速度更慢。

- Hadoop

  与 Hadoop 集群的总容量相比，Sqoop 使用的 Mapper 数量相对较少，所以这不太可能成为瓶颈。但是，最好还是核实一下，确定 Sqoop 的 Mapper 并没有等待任务槽可用，而且要检查磁盘 I/O、CPU 利用率以及数据节点的交换（Mapper 运行的数据节点）。

- 访问方式不良

  不存在主键时，或者输出查询结果时，用户需要指定自己的分片列。非常重要的是这个列要么是分区键，要么包含一个索引。多个 Mapper 对同一个表运行全表扫描，会导致 RDBMS 上明显的资源冲突。如果未发现该类分片列，那么输出数据时只能使用一个 Mapper。这种情况下，一个 Mapper 的运行速度其实要比多个含有较差的分片列的 Mapper 快。

6. 保证Hadoop的数据同步

数据从 RDBMS 采集到 Hadoop，这通常不是一个单一事件。随着时间的流逝，RDBMS 中的数据会发生改变，那么就需要更新 Hadoop 中的数据。一定要记住，在这种情况下，HDFS 为只读型的文件系统，不能更新数据文件（支持添加操作的 HBase 表除外）。想要更新数据，我们需要替换数据集、增加分区，或者通过合并变更创建新的数据集。

如果表相对较小而且采集整个表花费的时间较少，则不需要追踪表的修改或者添加。需要更新 Hadoop 中的数据时，可以简单地重新运行原始的 Sqoop 输出命令，然后采集整个表（其中包括了所有的修改），用新的版本取代旧的版本。

如果表比较大而且采集时间比较长，我们更倾向于只采集上次采集之后对表作出的修改。这里要求鉴别此类修改。Sqoop 支持以两种方法鉴别新行或者经过更新的行。

- Sequence ID

  如果每个行都有一个特定的 ID，而且新行的 ID 比旧行更高，那么 Sqoop 能够跟踪最后写入 HDFS 中的 ID。那么，下一次运行 Sqoop 时就只输出比上次 ID 更高的行。这种方法适用于事实表。事实表中添加了新数据，但是没有更新已经存在的行。

  比如，首先创建一个任务：

  ```
  sqoop job --create movie_id --import --connect \
  jdbc:mysql://localhost/movielens \
  --username training --password training \
  --table movie --check-column id --incremental append
  ```

  然后，增量更新 RDBMS 数据，执行该任务：

  ```
  sqoop job --exec movie_id
  ```

- 时间戳

  如果每个行都有一个时间戳来表示行的创建时间，或者上一次更新时间，那么 Sqoop 就能存储上次写入 HDFS 时的时间戳。执行下一个任务时，Sqoop 将只输出带有更高时间戳的行。这种方法适用于维度变化较慢的情况。这种情况既增加了行又更新了行。

运行 Sqoop 并开启 `--incremental flag` 开关时，你可以复用同一目录名，这样会把新的数据作为另外的文件存储到这一目录中。这意味着在这一数据目录上运行的任务不需要额外的操作就能看到所有即将到来的新数据。该设计的缺点是缺乏数据一致性：当 Sqoop 正在运行的时候，任务也开始执行，那么该任务可能只处理了部分已加载的数据。我们推荐让 Sqoop 将增量的数据更新加载到新的目录中。一旦 Sqoop 完成数据的加载（_SUCCESS 文件出现说明加载完成），就可以对数据进行清理和预处理，这些可以在将数据复制到数据目录之前进行。新的目录也可以作为新的分区添加到已存在的 Hive 表中。

当增量采集的数据不仅包含新的行，而且包含对已存在行的更新时，我们需要合并新数据集和旧数据集。Sqoop 支持这种操作，命令为 `sqoop-merge`。这里涉及的参数包括：合并键（通常为表的主键）、新旧目录以及作为参数的目标位置。Sqoop 将读取两个数据集，而当两行的键相同时，将保留最新的版本。`sqoop-merge` 的代码是相当通用的，支持 Sqoop 产出的任意数据集。如果数据集有序，并且进行了分区处理，合并处理操作可以利用这一点，以一种更高效的方式运行，即只包含 Map 的任务。

2.2.4 Flume：基于事件的数据收集及处理

Flume 是一种分布式的可靠开源系统，用于流数据的高效收集、聚集和移动。Flume 通常用于移动日志数据，但是也能移动大量事件数据，如社交媒体订阅、消息队列事件或者网络流量数据。我们将在这里简单地介绍一下 Flume，然后讨论使用时需要考虑的因素与建议。如欲了解更多关于 Flume 的内容，请参阅 Flume 官方文档（http://flume.apache.org/documentation.html）及《Flume：构建高可用、可扩展的海量日志采集系统》。

1. Flume架构

图 2-7 展示了 Flume 的主要组件。

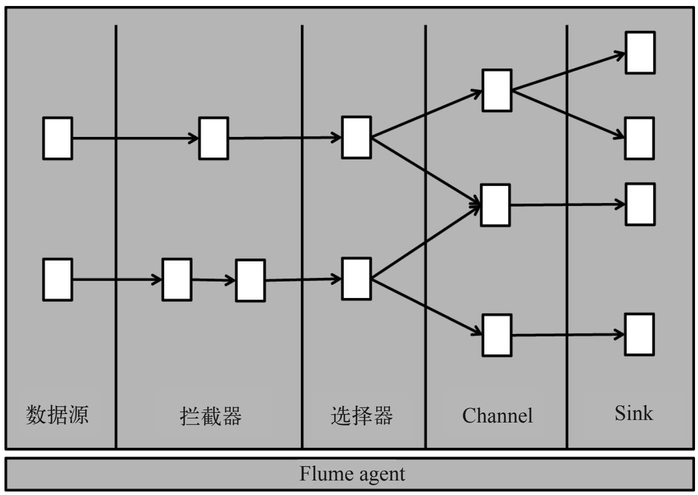


图 2-7：Flume 组件

- Flume 的数据源使用来自外部数据源的事件，然后转发到 Channel 中。外部数据源可以是任何一个能够产生事件的系统，比如 Twitter 这样的社交媒体网站、机器日志，或者消息队列。实施 Flume 数据源的目的是使用来源于特定外部数据源的事件，很多数据源都能与 Flume 一起使用，包括 AvroSource、SpoolDirectorySource、HTTPSource 与 JMSSource。
- Flume 拦截器能够拦截事件，并且能在传输过程中对事件作出修改。Flume 拦截器还能够转化事件、丰富事件或者实施Java类任何一种基本的操作。拦截器常用于格式化、分区、过滤、分片、验证，或者将源数据用于事件。
- 选择器为事件提供了路径。用户能够使用选择器将事件发送到零至多个路径。正因如此，如果需要分至多个 Channel，或者需要基于事件发送到特定 Channel，那么选择器会非常有用。
- Flume Channel 存储事件，直到填满一个 Sink。最常用的 Channel 为 Memory Channel 与 File Channel。Memory Channel 将事件存储于内存，在 Channel 之中提供了最佳性能。但是如果处理或者主机操作失灵，将会丢失事件，导致可靠性降到最低。更为常用的磁

盘 Channel 通过磁盘的持久存储提供更持久的事件存储。选择正确的 Channel 是一个很重要的构架决策，需要平衡性能与持久性。

- Sink 将事件从 Channel 中移除并传输到目的位置。目的位置可能是事件的最终目标系统，或者可以进一步进行 Flume 处理的位置。常用的 Flume Sink 是 HDFS Sink，顾名思义，它会将事件写入 HDFS 文件中。
- Flume agent 是这些组件的容器，承载着 Flume 数据源、Sink、Channel 等 JVM 进程。

Flume 的特点如下。

- *可靠性*

  事件会一直在 Channel 中存储，直到传输到下一个阶段。

- *可恢复性*

  事件可以持久化到硬盘，然后在出现错误时恢复。

- *声明式*

  无需编码，配置会指定各组件的组合方式。

- *高度定制化*

  尽管 Flume 包含大量的数据源、Sink 以及框架外的组成，但它提供高度可插拔的框架，能按照用户的需求定制化地实现。

关于保证的说明

尽管 Flume 提供了“有担保的”传输，但实际使用时并不能提供百分之百的保证。各种错误都有可能发生：内存错误、磁盘错误，甚至是整个节点都出现错误。虽然 Flume 不能为事件传输提供百分之百的保证，但是它可以通过配置调整安全指数。高水平的保证当然会导致性能降低。架构师的责任是决定需要哪个水平的保证，让保证与性能之间达到平衡。本章将进一步讨论相关的考虑因素。

2. Flume范式

以下模式阐明了 Flume 采集数据时的一些常见应用。

- *扇入*

  图 2-8 为一个扇入式架构的例子，这可能是最常见的 Flume 架构。本例中，每个数据源系统（即网络服务器）上都部署了一个 Flume agent，将事件发送到 Hadoop 边缘节点上的 agent。

  这些边缘节点应该在同一个网络中，与 Hadoop 集群的位置相同。使用多个边缘节点能够提供可靠性：一个边缘节点出现错误，事件不会丢失。我们也推荐在发送之前将事件压缩，这样能够减少网络流量。如果安全性很重要，也可以使用 SSL 来加密数据。在后面的点击流处理问题中，我们会看到 Flume 扇入式配置的完整例子。

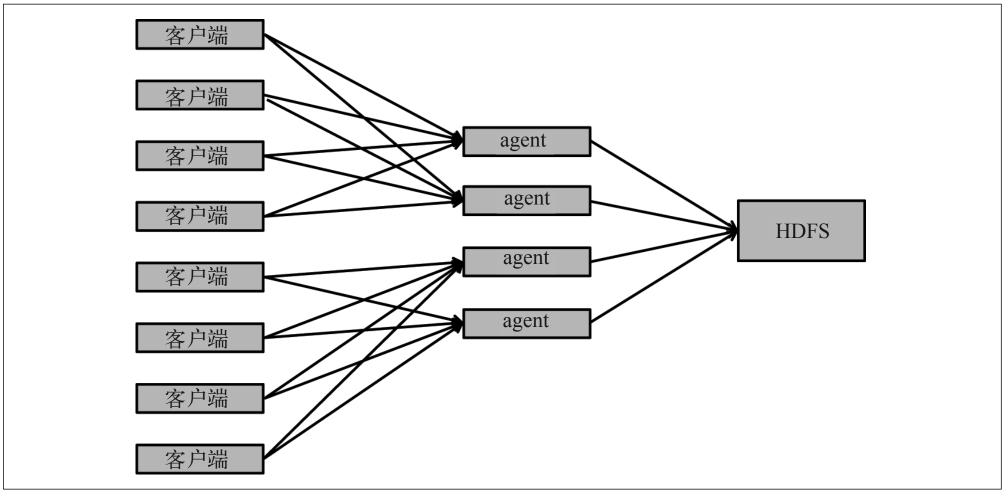


图 2-8：Flume 扇入式架构

- 采集时数据分片

  另一个常见的模式为将事件分片，然后采集到多个目标位置。这种模式经常用于将事件发送到主要集群，以及用于灾难恢复（Disaster Recovery，DR）的备用集群。图 2-9 中，我们使用 Flume 将同样的事件写入 Hadoop 的主要集群与备用集群。在主要集群无法使用的情况下，DR 集群更像事件中的错误恢复集群。有效备份 Hadoop 量级的数据集通常需要另一个 Hadoop 集群，所以该集群也可以用于数据备份。

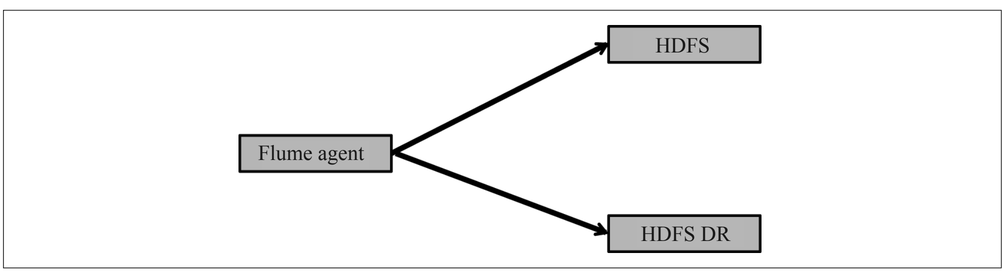


图 2-9：Flume 事件分片架构

- 采集时数据分区

  Flume 也能用于对采集的数据分区，如图 2-10 所示。比如，HDFS Sink 能按照时间戳对事件分区。

- 流式数据分析时的事件分片

  到目前为止，我们只讨论了最终目标为持久层的 Flume，但是 Flume 也常用于将事件发送到一个流分析引擎，如 Storm 或 Spark Streaming，该类引擎能够生成实时计数（Real-time Counts）、窗口（Windowing）与汇总（Summaries）。就 Spark Streaming 而言，由于它实现了 Flume 的 Avro 数据源接口，所以集成起来很简单。因此，我们只需要将一个 Flume Avro Sink 指向 Spark Streaming 的 Flume Stream 即可。

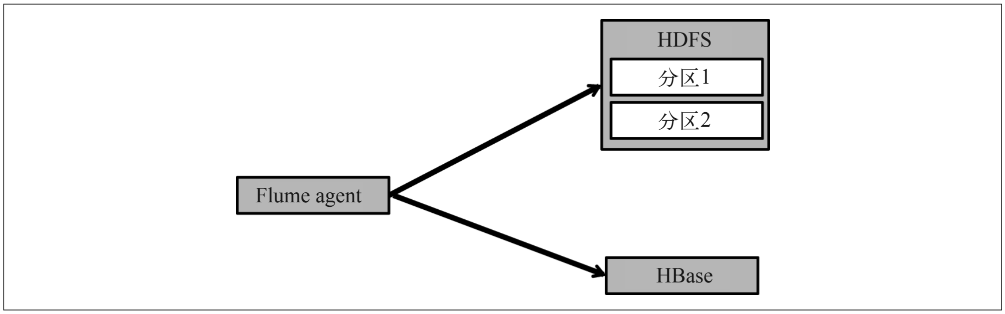


图 2-10：Flume 分区式架构

3. 文件格式

使用 Flume 将数据采集到 HDFS 时，需要考虑以下文件格式方面的问题（第 1 章已经深入讨论了存储方面需要考虑的问题）。

- 文本文件

  使用 Flume 处理事件时，文本是一个非常常见的格式，但是不是 HDFS 文件的最佳格式，原因如第 1 章所述。一般来说，通过 Flume 采集的数据应该保存在 SequenceFile（HDFS Sink 的默认格式）中，或者作为 Avro 进行保存。注意，将文件保存为 Avro 时，采集和后续处理可能需要额外的工作转化数据。我们将提供一个点击流处理中的 Avro 保存案例。SequenceFile 与 Avro 都能提供有效压缩，也都是可分片式的。Avro 为优先选项，因为它能够将模式作为文件的一部分进行存储，也能提供更有效的压缩。Avro 格式内部有检查点，如果写入文件时出现问题，Avro 能提供更好的错误处理。

- 列式文件格式

  列式格式（如 RCFile、ORC 或 Parquet）也不适用于 Flume。这些格式的压缩更有效，但与 Flume 一起使用时，需要批量的数据，这就表明如果出现问题可能会丢失更多数据。另外，Parquet 需要在文件尾部写入模式，所以一旦出现问题，整个文件就会全部丢失。

Flume 事件序列化器（Flume Event Serializer）将一个 Flume 事件转化为另一种格式后输出，所以能通过它写入不同格式的事件。Flume HDFS Sink 支持文本格式和 SequenceFiles 格式，所以可以直接将文本格式或者 SequenceFiles 格式的事件写入 HDFS。我们在前面提到过，采集 Avro 格式的文件时，采集和后续处理阶段可能需要一些额外的工作。注意，Avro 的事件序列化器可用于 HDFS Sink，但是创建 Avro 数据时将使用事件模式，而事件模式并不是存储数据的首选模式。不过也可以按照用户自己的逻辑需求重写 EventSerializer 接口，而后创建自定义的输出模式。持久性数据经常需要创建格式，而且格式不能与发送的 Flume 事件相同，所以创建自定义的事件序列化器是一种常见的任务。注意，拦截器也可用于此类格式化，但是事件序列化器离磁盘最后的输入位置更近，然而在到达序列化阶段之前，还需要通过一个 Channel 与一个 Sink。

4. 推荐使用方式

以下为使用 Flume 的一些意见以及最佳实践。

Flume 数据源 Flume 数据源当然是 Flume 流水线的开始部分。Flume 数据源是数据（比如，从 Avro 数据源）输入 Flume，或数据输出到其他系统（比如 JMS 数据源）的位置。配置 Flume 数据源时需要考虑两个主要因素，这也是 Flume 能达到最佳性能的关键因素。

- 批处理大小

  Flume 批中事件的数量能对性能造成极大的影响。以 Avro 数据源为例，用户将一批事件发送到 Avro 数据源，而后必须等待。直到事件进入 Channel，而且 Avro 数据源通过一个成功信号对用户做出反馈。这看起来似乎很简单，但是网络延迟会对该过程造成很大的影响；如图 2-11 所示，几毫秒的延迟都会显著拖延事件处理。图中，输出一批事件存在 12 毫秒的延迟。现在，如果要加载一百万个事件，设定批处理大小为 1，则传输时间为 3.33 小时；如果批处理大小为 1000，则传输时间为 12 秒。因此，配置 Flume 时应该设置一个合适的批处理大小，这样才能获得最优性能。首先将批处理大小设定为 1000，一次 1000 个事件，而后根据性能在此基础上上下调整。

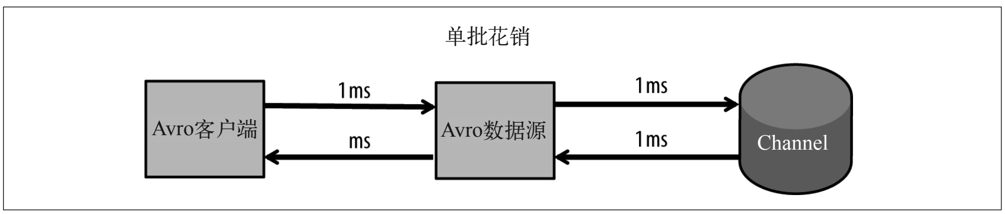


图 2-11：延迟对 Flume 性能的影响

- 线程数

  理解 Flume 数据源使用线程的方法，能够使用户最大化地利用包含数据源的多线程。一般来说，线程越多，网络或 CPU 的占用率就越高。Flume 中，不同的数据源线程也有所不同。比如，Avro 数据源是一种 Netty 服务[4]，而且在同一时间内用户可以多次连接该数据源，所以只需简单地增加更多用户或者用户线程，就能使 Avro 数据源多线程化。JMS 数据源则不同，它是一种主动拉取数据的数据源，所以如果想获得更多线程，就需要在 Flume agent 中配置更多数据源。

Flume Sink 记住，Sink 的作用是引导数据进入最终目的位置。在 Flume agent 中配置 Flume Sink 需要考虑以下因素。

- Sink 数

  一个 Sink 只能从一个 Channel 中抓取数据，但是多个 Sink 能从同一个 Channel 中抓取数据。一个 Sink 只能在单线程上运行，所以单个 Sink 存在局限性，比如，磁盘的吞吐量就会受到限制。假设对于 HDFS，一个磁盘占有的空间为 30 MBps，如果用户只有一个写入 HDFS 的 Sink，那么使用该 Sink 所能得到的最大吞吐量为 30 MBps。使用多个 Sink 从同一个 Channel 中抓取数据，将解决这一瓶颈问题。网络或 CPU 会对更多 Sink 的使用造成限制。除非集群非常小，否则 HDFS 不应该成为瓶颈。

注 4：Netty 是 Java 的一个软件框架，旨在简化客户端 – 服务器网络应用程序的开发。

- 批处理大小

  记住，在所有的磁盘操作中，缓存都为吞吐量带来了显著的优势。也要记住，对于保证 Channel 正常运行的 Flume Sink，提交事件后需要将事件存储到磁盘中。也就是说，输出到磁盘的数据流需要清空，进而需要在开始时调用系统命令 `fsync`。如果 Sink 获取了较小批的数据，那么将有大量时间花费大在这些系统命令的执行上。这里只有一个大缺陷：Sink 获取大批数据时增加了复制事件的风险。比如，事件写入 HDFS 之后 Flume 处理暂停，但是未被确认，那么当处理重新开始时，事件就会被重新写入。因此，很重要的一点就是谨慎调整批处理大小，这样才能获得吞吐量与潜在复制风险之间的平衡。但是，就像我们刚才所说的，增加一个后续处理步骤以移除复制记录会相对简单一些。我们随后看一下案例。

Flume 拦截器 拦截器是 Flume 中非常有用的组件。拦截器提供了强大的功能，能够获取一个或者一组事件，并且对事件进行修改、过滤或者分片。但是，仍需要警惕：一个自定义的拦截器当然使用自定义代码，而自定义代码却会带来一些风险，如内存泄露或 CPU 消耗过多。记住，作用越强，将 bug 引入生产环境的风险就越高，所以仔细检查和检测所有的自定义拦截器代码是非常重要的。

Flume Memory Channel 如果性能是主要考虑因素，而且数据丢失不是主要问题，那么最好选择 Memory Channel。记住，对于 Memory Channel，如果一个节点出现问题，或者控制 Channel 的 Java 处理被结束，那么仍然处于 Memory Channel 的事件将永久丢失。而且，能够存储的事件数量受可用的 RAM 限制，所以在下流出现错误时，Memory Channel 缓存事件的能力也会受到限制。

Flume File Channel 如前所述，由于 File Channel 使事件持久化地存储于磁盘上，所以 File Channel 比 Memory Channel 更持久耐用。但是，值得注意的是，File Channel 的配置可以在持久性与性能之间做出权衡。

在推荐使用 File Channel 之前需要注意，除了能将事件写入磁盘，File Channel 也能将事件写入周期性的检查点。这些检查点有助于重新启动和恢复 File Channel。所以，在配置 File Channel 确保可靠性时，应该考虑一下这些检查点。

以下为使用 File Channel 的一些考虑因素与建议。

- 可以配置 File Channel 使用多个磁盘，在不同磁盘之间交替写入事件。这样有助于提高性能。而且，如果一个节点包含多个 File Channel，那么应该使用明确的目录，并且让每个 Channel 对应不同的磁盘。
- 使用企业存储系统（如 NAS）能保证数据不会丢失，但是会降低性能，增加成本。
- 使用双重检查点目录。应该使用一个备份的检查点目录为防止数据丢失提供更高的保证。通过设定 `useDualCheckpoint` 为 `true` 来配置该目录，并为 `backupCheckpointDir` 设定对应的目录。

我们也应该注意相对而言不太常用的 JDBC Channel。JDBC Channel 能够将事件永久存储于兼容 JDBC 的数据库，如关系型数据库。这是持久性最好的 Channel，但是写入数据库的损耗很高，所以最不常用。出于性能方面的考虑，一般不推荐使用 JDBC Channel。只有在应用对可靠性要求极高的时候才应该使用 JDBC Channel。

关于 Channel 的说明

我们已经讨论了 Memory Channel 与 File Channel 之间的差异和各自的适用情况，但是在某些情况下，同一个架构中需要同时使用两种 Channel。一种常见的模式是，将事件分片，从一个持久性 Sink 发送到 HDFS，以及流数据 Sink。流数据 Sink 将事件发送到 Storm 或 Spark Streaming 之类的系统，然后进行数据流分析。持久性 Sink 为了达到持久性可能会使用 File Channel，但是对于数据流 Sink 来说，主要需求可能是实现最大吞吐量。另外，流数据输出的时候，可以接受事件丢失，而 Memory Channel 具有最好用、性能最佳的特点，所以最好的选择是 Memory Channel。

Sizing Channel 配置 Flume agent 时，经常会遇到是增加还是减少 Channel，或者如何设定 Channel 容量的问题。以下为一些简单的指导意见。

- Memory Channel

  在单一节点上，要限制 Memory Channel 的数量。不用说，单一节点上配置的 Memory Channel 越多，每个 Channel 可用的内存就越少。注意，一个 Memory Channel 能包含多个数据源，而且能被多个 Sink 抓取数据。Sink 速度通常比相应的数据源慢，或者相反，因此应当让单个 Channel 多连接较慢的组件。存在不同流水线时，每个节点上应该设置更多 Memory Channel，表明通过流水线输送的数据要么各不相同，要么被输送到不同的位置。

- File Channel

  当 File Channel 只支持将事件写入一个驱动器时，应该配置多个 File Channel，以利用更多磁盘。既然 File Channel 支持将事件写入多个驱动器，甚至是同一驱动器的多个文件，那么使用多个 File Channel 并不会带来性能优势。再说一遍，在一个节点上配置多个 Channel 的原因是存在不同的流水线。然而，对于 Memory Channel，配置 Channel 时也应该考虑一个节点上可用的内存量。至于 File Channel，当然需要考虑可用的磁盘空间。

- Channel 大小

  记住，Channel 是缓存区，不是存储区，它的主要功能是在数据全部输送到 Sink 之前存储数据。通常来讲，缓存区一定要足够大，这样 Sink 才能从 Channel 中抓取配置批量大小的数据。如果 Channel 达到了最大容量，那么就很可能需要更多的 Sink，或者需要通过更多节点进行写入。如果 Sink 的数量已经达到最大，那么 Channel 的容量变大不会有什么帮助。实际上，容量更大的 Channel 反而可能造成不良影响，比如，非常大的 Memory Channel 存在垃圾回收，结果会导致整个 agent 的速度变慢。

5. 定位Flume瓶颈

Flume 有很多配置方法。因为选择太多，所以开始时可能会应接不暇。但是，之所以存在这么多选择，原因在于 Flume 流水线是基于各种网络和设备而创建的。对于 Flume 的性能问题，这里列出了需要考虑的几点因素。

- 节点间延迟

  批量数据需要在网络中产生一个来回，才能从客户端提交到数据源。如果一批的处理量很小，那么较高网络延迟会对性能造成不良影响。使用多线程或者较大的批处理量有助于缓解该问题。

- 节点间吞吐量

  通过网络传输的数据量存在一定的限制。无法增加网络带宽时，可以考虑使用 Flume 的压缩来增加吞吐。

- 线程数

  使用 Flume 时，某些情况可以利用多线程提高性能。一些 Flume 数据源支持多线程，在其他一些情况下，可以通过配置 agent 来增加多个数据源。

- Sink 数

  HDFS 由多个驱动器组成。一个 HDFS Sink 一次只能写入一个磁头（spindle）。如先前所述，使用多个 Sink 写入数据能够提高性能。

- Channel

  如果使用 File Channel，则应该了解写入磁盘的性能限制。我们也讲过，一个 File Channel 可以写入多个驱动器，因此有助于提高性能。

- 垃圾回收问题

  使用 Memory Channel 时，可能会遇到垃圾回收的问题。当事件在 Channel 中的存储时间过长时，这种情况可能会出现。

2.2.5 Kafka

Apache Kafka 是一种发布–订阅消息的分布式系统，能够将消息归类为不同主题。应用程序能在 Kafka 上发布信息，或订阅主题进而接收特定主题下发布的消息。Producer 发布消息，而 Consumer 收集并处理消息。作为分布式系统，Kafka 在集群中运行，每个节点被称为 Broker。

Kafka 维护每个主题的分区日志。消息会发布到相应的主题中，每个分区都是一个有序的消息子集。同一个主题的多个分区能够通过集群中的多个 Broker 传送，这种方法提高了主题的容量与吞吐量，使其超越了单一机器所能提供的容量与吞吐量。消息在分区内被有序排列，每个消息都包含一个特定的偏移量。Kafka 中消息可以通过一个包含主题、分区以及偏移量的组合来确定。Producer 能够根据消息的主键选择消息应该写入哪一个分区，也能够简单地用循环的方式，让消息分布在各分区之间。

Consumer 会在 Consumer 组中注册，每个组包括一个或多个 Consumer，每个 Consumer 读取一个或多个主题分区。每组中的每条消息只能传送给一个 Consumer。但是，如果多个组订阅了同一个主题，那么每个组都将得到所有的消息。一个组中包含多个 Consumer 有助于获得加载平衡（可以支持高于单个 Consumer 处理能力的吞吐量）与高可用性（如果一个 Consumer 出现错误，它所读取的分区将重新分配给组中其他 Consumer）。

前面已经谈到，对于应用层面的数据分类，主要单位是主题。一个 Consumer 或 Consumer 组将读取其订阅主题的所有数据，所以如果一个应用只关注一个数据子集，那么就应该将该数据子集与其他数据放在两个不同的主题中。如果多个信息集总是一起读取和处理，那么应该将它们归在同一个主题中。

不过，分区是并行化处理的主要的单元。每个分区只能配对一个服务器，但是一个主题可以与分区总和同样大。另外，每个分区的信息最多由同一组中的一个 Consumer 读取，所以尽管可以通过增加 Consumer 数量来增加读取吞吐量，但其实主题中可用分区的数量会带来限制。因此，我们建议每个节点上分区的数量至少与集群中服务器的数量一样多，为以后几年的增长作出准备。实际上，每个主题可以包含几百个分区，这里并不存在任何缺陷。

Kafka 存储了预先配置时长范围内（通常为几周或几个月）的所有消息，而且对于每个 Consumer 读取的信息只保留最后一条消息的偏移量。因此，用户可以从最近一次正确的偏移量开始，重新读取主题分区，进而从错误中恢复。用户也可以将消息队列回溯，然后重新读取信息。在解决 bug 以及其他问题时，“回溯”特点会非常有用。如果只存储一段时间内的所有消息，不跟踪记录每个 Consumer 和消息的确认，那么 Kafka 能够扩展到 10000 多个 Consumer，支持它们的非频繁批量读（如 MapReduce 任务），甚至是在吞吐量非常高的情况下，也能保持较短的延迟。传统的信息 Broker 则跟踪用户的确认消息，通常需要在内存中存储所有未收到答复的信息。如果用户数量比较多，或者读取批数据的用户数量比较多，就会导致交换，使性能严重降低。

以下为 Kafka 的常见用途。

- Kafka 能取代应用架构中的传统信息 Broker 或者信息队列，用于分离服务。
- Kafka 最常用于高速活动流，如网站点击流（website clickstream）、度量（metrics）以及日志（logging）。
- Kafka 也常用于流数据处理。它可以同时用作信息流的来源和输送目的位置（数据流任务在目的位置记录其他系统读取的结果）。第 7 章会讲解相关的例子。

1. Kafka对容错的支持

每个主题分区都能复制到多个 Broker，进而获得连续的服务，而且在 Broker 出现错误时不会丢失数据。对于每个分区，一个复制分区被设定为 leader，而其他复制分区被设定为 follower。leader 完成所有的读写操作。follower 作为保险备份使用：如果 leader 出现错误，那么将从其他同步的复制分区中选择一个作为新的 leader。如果一个 follower 的复制操作与 leader 的复制操作相隔时间不是太久，那么就可以认为该 follower 与 leader 同步。

写入数据时，Producer 可以选择让所有的同步分区都反馈写入操作的确认消息，或者是只让 leader 反馈，或者根本不等待确认消息。Producer 等待的确认消息越多，就越能保证在出现错误时不丢失写入的数据。但是，如果可靠性不是主要考虑因素，那么等待的确认消息越少，获得的吞吐量就越高。管理者也能设定同步复制分区数量的最小阈值——如果低于该限度，要求所有同步复制分区都反馈确认消息的写入操作将被拒绝。这种方法提供了额外的保证，因为信息不会被意外地只写入单一的 Broker，这种情况下如果 Broker 出现错误，数据可能会丢失。

注意，如果网络在消息发送之后、收到确认消息之前出现错误，那么 Producer 无法检查消息是否已经写入 Broker。Producer 需要决定是冒着重复的风险重新发送消息，还是冒着丢失数据的风险跳过。

读取数据时，用户只能读取*提交的消息*（committed message），也就是写入所有复制分区的消息。这表明用户不必担心意外读取会在 Broker 出现错误后消失的数据。

Kafka 保证 Producer 发送到特定主题分区的所有消息，都会按照发送顺序写入分区。用户看到的消息会按照发送顺序排列。如果用复制因子 N 来定义一个分区，那么 N-1 个节点的一个错误不会引起任何数据的丢失。

根据用户的配置，Kafka 能够支持至少一次、*至多一次*与有且只有一次的传输语义。

- 如果 Consumer 在处理完消息之后再增加当前的偏移量（默认设置），那么一个 Consumer 可能会在数据处理之后、增加偏移量之前出现错误。这样一来，一个新的 Consumer 将从最近一次的偏移量处开始重新读取消息，消息*至少会被处理一次*，但是可能出现重复。
- 如果 Consumer 先增加偏移量，再处理消息，那么 Consumer 可能会在增加偏移量之后、处理数据之前出现错误。这种情况下，新的 Consumer 将从新的偏移量中读数据，跳过已经被读取但是未出现错误的 Consumer 处理过的消息。信息*最多被处理一次*，但是可能会丢失数据。
- 如果 Consumer 使用一个两阶段的事务来保证处理数据和增加偏移量同时发生，信息将*仅处理一次*。两阶段的事务性能损耗较大，所以最好在批处理类型的 Consumer 中而不是流式处理时使用。Camus 是一个随后将要讨论的 Hadoop Consumer，包含了有且只有一次处理的语义——通过 `Mapper` 同时将数据与偏移量写入 HDFS 中，如果 `Mapper` 出现错误，两者将同时移除。

回顾 Kafka 的高可用性（High Availability，HA）保证，我们还需要再明确一点，即可以在多个数据中心使用 Kafka。这种部署方式将一个集群作为另一个集群的 Consumer。正是这种消息复制机制保证了所有 Consumer 的高可用性。

2. Kafka与Hadoop

Kafka 是一种通用的信息 Broker，并不专门用于将数据写入 Hadoop。但是，有一些用法合并了两种系统。一种用法涉及从 Kafka 中提取数据，并且将数据存储到 HDFS 中，用于离线处理。比如，追踪网站上的点节流活动时，用户可以使用 Kafka 发布站点活动，如网页浏览与搜索。实时仪表板（dashboard）或者实时监控器（monitor）可以获取这些信息，同样存储到 HDFS 中，并用于离线分析与报告。

另一个常用的用法是实时流数据处理。Kafka 作为数据源，为 Spark Streaming 或 Storm 提供一个可靠的信息流（详见第 7 章）。

一个常见的问题是，要不要使用 Kafka 或 Flume 将日志数据或其他流数据源采集到 Hadoop 中。跟一贯的情形一样，这主要取决于需求。

- Flume 是一种更完整的 Hadoop 数据采集方法，提供了良好的支持，可以将数据写入 Hadoop 中（包括 HDFS、HBase 与 Solr）。Flume 是基于配置设定的，所以 Hadoop 管理

员能够部署和使用 Flume，而无需编写任何代码。将数据写入 HDFS 时，Flume 能处理很多常见的问题，如可靠性、最佳文件大小、文件格式、更新源数据以及分区。

- Kafka 是一种具备高可用性与高性能的可靠数据源。如果要求涉及容错信息的传输、信息重放或者大量的潜在 Consumer，那么 Kafka 会成为极佳的选择。但是，这也表明用户将开发自己的 Producer 与 Consumer，而不是使用已经存在的数据源与 Sink。

比较来看，Flume 与 Kafka 似乎是互补的。Kafka 提供一种具有高吞吐量的灵活可靠输出，而 Flume 提供使用方法更简单的数据源、拦截器以及 Hadoop 集成。因此同时使用两者效果更好。

Flume 包括一个 Kafka 数据源、Kafka Sink 以及 Kafka Channel。这些能够将事件从 Kafka 发送到 Flume Sink（如 HDFS、HBase 与 Solr），或者从 Flume 数据源（如 Log4j 与 Netcat）发送到 Kafka。你甚至可以使用 Kafka，通过可靠的 Channel 来加强 Flume 的功能。

Flume 的 Kafka 数据源是一种 Kafka Consumer，能够从 Kafka 中读取数据，并发送到 Flume Channel 中。接下来，数据通过 Flumed 拓扑结构继续传输。配置数据源很简单，只需设置主题、Kafka 使用的 ZooKeeper 服务器以及 Channel。数据源允许调整发送到 Channel 的批数据的大小。使用较小的批处理量，延迟更低；使用较大的批处理量，吞吐量更高，而且降低了 CPU 的使用率。

从 Kafka 中读取数据时，Flume 默认使用 groupId `flume`。向同一个 groupId 增加多个 Flume 数据源，每个 Flume agent 都将得到一个信息子集，而且能够增加吞吐量。另外，如果其中一个数据源出现错误，那么其他数据源将重新调整，所以能够继续采集信息。Flume 的 Kafka 数据源是可靠的。数据源、Channel、Sink 或者 agent 出现错误，数据并不会丢失。

Flume 的 Kafka Sink 是一个 Kafka 的 Producer，会将数据从 Flume Channel 发送到 Kafka。配置 Kafka Sink 需要设置主题以及 Kafka Broker 的列表。与 Kafka Source 类似，调整 Kafka Sink 的批处理大小可以提高吞吐量或降低延迟。

Flume 的 Kafka Channel 合并了 Producer 和 Consumer。当 Flume 数据源将信息发送到 Kafka Channel 时，这些事件会发送到 Kafka 主题中。每个批处理数据都会发送到不同的 Kafka 分区，所以写入操作将达到负载平衡。当 Kafka Sink 从 Channel 中读取数据时，事件将由 Kafka 进行采集。Kafka Channel 是高度可靠的。当服务器出现错误时，只要 Kafka Channel 与一个配置正确的 Kafka 服务器一起使用，就能够保证不丢失信息，甚至不会引起显著延迟。

流数据在这里并不适用。批处理通常能够提高吞吐量，而且更适合压缩的列式存储，如 Parquet。如果要从 Kafka 中批量加载数据，我们推荐使用 Camus（http://github.com/linkedin/camus）。这是一种独立的开放性数据源项目，能够将数据从 Kafka 中采集到 HDFS 中。Camus 使用起来很灵活，而且相对来说容易维护。Camus 的特点是可以从 ZooKeeper 中自动查找 Kafka 主题，将数据转化为 Avro 与 Avro 模式管理，以及自动分区。在任务设置阶段，Camus 提取主题列表与开始时信息的 ID，进而提取 ZooKeeper 中的数据，然后将 Map 任务中的主题分片。每个 Map 任务都从 Kafka 中提取信息，按照时间戳将信息写入 HDFS 目录。当任务成功结束时，Camus Map 任务只能将信息转移到最终输出位置。因此，这里可以使用 Hadoop 的推断执行，而不存在将信息重复写入 HDFS 的风险。

图 2-12 是一个整体框图，能帮助你理解 Camus 的执行方式。

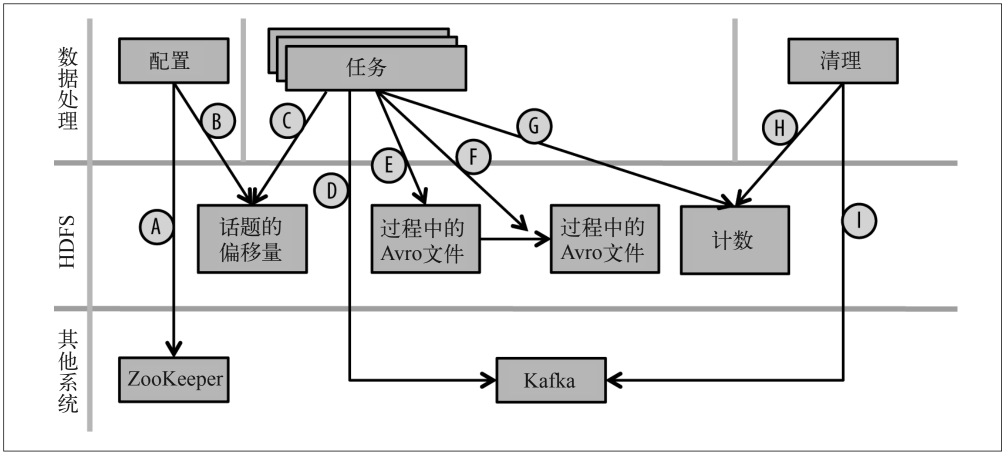


图 2-12：Camus 执行数据流图

图中主要包括以下几步。

- A：设置步骤时从 ZooKeeper 中提取 Broker URL 以及主题信息。
- B：设置步骤时将主题与偏移量信息持久性地存储到 HDFS 中，以供任务读取。
- C：任务读取设置步骤时的持久化信息。
- D：任务从 Kafka 中采集事件。
- E：任务将数据写入 HDFS 中的临时位置，并由用户自定义的解码器定义数据的存储格式，这里用的是 Avro 文件格式。
- F：当任务开始清理时，将数据从临时位置转移到最终位置。
- G：任务输出相关活动信息的审计计数。
- H：作为清理阶段，读取所有任务的审计计数。
- I：作为清理阶段，将持久化的报告返还 Kafka。

如果想要使用 Camus，你可能需要自己编写解码器，将 Kafka 信息转化为 Avro。这与 Flume HDFSEventSink 中的序列化器相同。

2.3　数据导出

本章主要关注数据如何导入 Hadoop，在 Hadoop 上设计应用时，这块内容通常会花费更多的时间。当然，将数据导出 Hadoop 也很重要，而且数据导入 Hadoop 时要考虑的因素也适用于此。以下是从 Hadoop 导出数据的一些常见场景与考虑因素。

- *将数据从 Hadoop 导出到 RDBMS 或数据仓库*

  一个常见的模式是使用 Hadoop 将传输的数据采集到数据仓中——换句话说，使用 Hadoop 进行 ETL。大多数情况下，传输的数据都可以使用 Sqoop 采集到目标数据库中。但是，如果不选择 Sqoop，也可以先将文件简单地从 Hadoop 中提取出来，然后使用特

定于供应商的采集工具采集。使用 Sqoop 时应该尽量使用特定于数据库的连接器。无论使用哪种方法，都应该避免数据路的过量加载。在 Hadoop 中容易管理的数据容量在传统数据库中可能并不容易管理。给自己和数据库管理员行个方便，一定要认真考虑目标系统的加载情况。另外，当 Hadoop 发展成熟而且弥补了与传统数据库管理系统之间的容量差距时，我们会看到越来越多的数据从这些传统数据库转移到 Hadoop 中，而不必将数据从 Hadoop 中导出。

- *导出数据供外部程序分析*

  与上一点相同，Hadoop 是一种很强大的工具，能够处理和汇总数据，然后将数据输入外部分析系统和应用中。这些情况更适合使用简单的文件传输，比如使用 `hadoop fs -get` 命令或者可挂载的 HDFS 方法。

- *在 Hadoop 集群间移动数据*

  在 Hadoop 集群之间转移数据是很常见的，例如为了错误恢复或在多个集群之间移动数据。这种情况下，可以使用 DistCp 为 Hadoop 集群之间的数据传输提供简单有效的方法。DistCp 使用 MapReduce 来执行大量数据的并行化传输。数据源或者目标位置不是 HDFS 文件系统时，也可以使用 DistCp。举个例子，将数据转移到云系统 [比如 Amazon 的 Simple Storage System（S3）] 就是一个越来越常见的需求。

2.4 小结

现在，似乎有越来越多的工具可以将数据输入或输出 Hadoop，但是就像本章讲到的，如果仔细考虑自己的需求，那么找到正确的方法并不是很困难。其中一些考虑因素如下。

- 数据源系统（从该系统将数据采集到 Hadoop 中）或者目标系统（从 Hadoop 中提取数据到其中）。
- 采集或提取数据的频率。
- 采集或提取的数据类型。
- 数据处理或评估的方式。

理解可用工具的作用与限制以及自己的特定需求，你就可以设计出可伸缩的稳定架构。

第 3 章

Hadoop数据处理

前面的章节里讨论了 Hadoop 数据建模方面需要考虑的若干问题，以及如何在 Hadoop 上导入和导出数据。数据导入 Hadoop，完成数据建模之后，我们自然希望能够访问和利用这些数据。本章会探讨 Hadoop 上已有的数据处理框架。

和 Hadoop 的其他问题一样，在数据处理上，我们需要先了解可以选择的框架，然后再确定使用哪一种。了解这些选择，我们才能清楚地知道如何挑出最适合任务的工具，不过这也会让新手感到困惑。本章的目的就是帮助你基于自己的实际场景做出正确的选择。

本章会先介绍主要的执行引擎，也就是在 Hadoop 集群上直接执行数据处理任务的框架。这包括成熟的 MapReduce 框架，以及新一代数据流引擎，如 Spark。

接下来，我们会介绍更高一层的抽象工具，如 Hive、Pig、Crunch 和 Cascading。相比更低一层的框架（如 MapReduce）而言，这些工具的设计目标是提供更易用的抽象接口。

对于每一个数据处理框架，我们都会谈到以下几个方面：

- 框架概述
- 框架使用示例
- 框架使用场景描述
- 关于该框架的更多资源介绍

通过阅读本章，你可以了解几种数据处理框架，但无法做到精通。本章的目标是让读者可以自信地为实际场景选择合适的工具。如果想了解更多细节，我们会提供深入研究特定工具的相关资料。

无共享架构

在了解特定框架之前，你需要注意这些框架的共同点：它们会尽可能地实现成一个无共享的架构。在分布式系统中，无共享架构是指系统中的每一个节点完全独立于其他任何节点，这样就不会在共享资源上存在瓶颈。无共享资源中的资源是指物理层次的资源，如内存、磁盘和 CPU 等。Hadoop 的处理框架使用的是分布式的 HDFS 存储，而不是集中式存储。无共享资源也指无共享数据。在这类框架中，每一个节点处理数据的一个确定的子集，而不需要访问共享的数据。无共享架构具有良好的扩展性。因为无共享资源，所以新加入的节点就能够扩增系统的资源，并且不会引发资源竞争。这些框架也具有故障包容的特点，每个节点彼此独立，因此不存在单节点故障。某节点发生故障后，系统能够快速恢复。读者在阅读本章时，可以注意到，每个框架在细节上有诸多不同，但有一点是类似的：它们都遵从了无共享架构的设计原则。

3.1 MapReduce

MapReduce 模型，最早出现在 Google 公司 Jeffrey Dean 和 Sanjay Ghemawat 的论文“MapReduce: Simplified Data Processing on Large Clusters”（http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf）中。这篇论文描述了一种用于海量数据处理和生成的编程模型及其实现。该编程模型提供了一种并行处理大数据集的应用开发方式，减少了许多开发分布式并发应用时常常会遇到的编程问题。该模型描述的无共享架构提供了一种高扩展性的系统实现方式，并对单点故障或进程失败提供了容错机制。

3.1.1 MapReduce概述

MapReduce 编程范式将数据处理拆分成了两个基本的阶段：Map 阶段与 Reduce 阶段。每个阶段的输入和输出均为键值对。

Map 阶段对应的进程称为 `Mapper`。`Mapper` 是（JVM 上运行的）Java 进程，通常在要处理的数据所在的节点上启动。利用数据的本地性是 MapReduce 的一个很重要的基本原则。对于大数据集而言，将处理进程分配至包含数据的节点上，比通过网络传输数据高效得多。在 `Mapper` 中进行的数据处理类型，通常有解析、变换和过滤。当 `Mapper` 处理完输入数据之后，会给下一阶段（Sort & Shuffle）产生键值对。

在 Sort & Shuffle 阶段，数据会排序和分区。本章会在后面详细介绍相关内容。分区后的有序数据将会通过网络发送给 `Reducer` 所在的 JVM，`Reducer` 会读取这些按照键进行分区的有序数据。`Reducer` 拿到这些记录，`reduce()` 函数就可以对数据执行各种操作了。不过，`Reducer` 通常会写出部分数据，或者聚合到 HDFS 或 HBase 的存储中。

总结来说，JVM 有两类，一类读取无序的数据，一类处理分区后的有序数据。我们稍后会讲述 MapReduce 的其他相关内容，图 3-1 展示的是我们刚才描述过的内容。

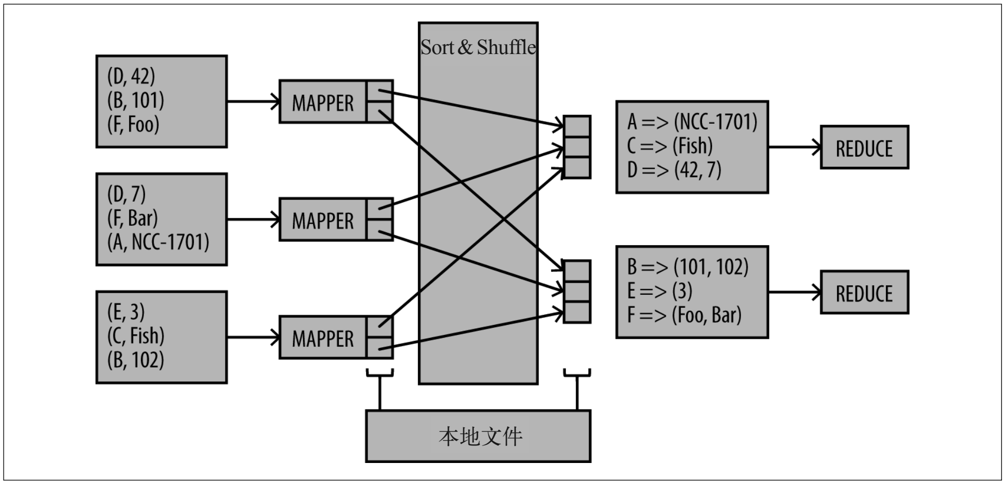


图 3-1：MapReduce 的 Sort & Shuffle

下面介绍 MapReduce 处理的典型特点。

- `Mapper` 以键值对的形式处理输入数据，并且每次只能处理一个键值对。`Mapper` 的个数是由框架而非开发人员决定的。
- `Mapper` 将键值对作为输出传递给 `Reducer`，但是不能将数据转发给其他的 `Mapper`。`Reducer` 同样不能与其他的 `Reducer` 通信。
- `Mapper` 和 Reducer 通常不会使用太多的内存，一般会将 JVM 的堆大小设置为相对较小的值。
- 虽然并非总是如此，但是一般来讲，每一个 `Reducer` 都会有单个输出数据流。在默认情况下，这是一个文件合集，命名形式类似于 part-r-00000、part-r-00001 等，存在单个 HDFS 目录下。
- `Mapper` 与 `Reducer` 的输出均会写入磁盘。如果 `Reducer` 输出的结果仍需额外的处理，那么整个数据集将会写入磁盘中，再读取一遍。这种模式被称作*同步屏障*（synchronization barrier）。这也是使用 MapReduce 迭代式数据处理比较低效的主要原因。

在深入探讨 MapReduce 的底层细节之前，还有一个要点需要注意：MapReduce 有两大弱点，使它不适用于迭代算法。其一是启动时间较长。即使 MapReduce 过程几乎什么都不做，启动也需要大概 10~30 秒。其二是 MapReduce 会频繁写入磁盘，以便容错。本章后面会介绍 Spark，我们会发现这些磁盘 I/O 并不是必需的。如图 3-2 所示，在一般的数据处理过程中，MapReduce 的确进行了多次的磁盘读写操作。

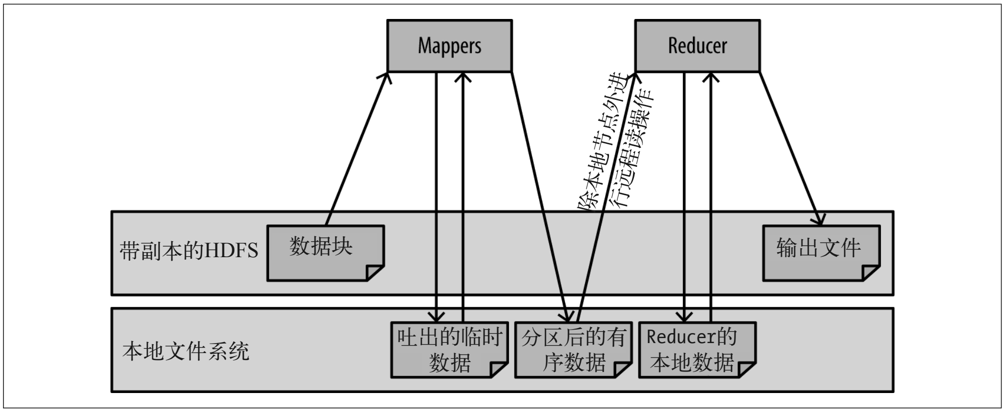


图 3-2：MapReduce I/O

MapReduce 的一个强大之处在于，它不仅仅是由 Map 和 Reduce 任务构成的，还包括协同工作的多个组件。每个组件均可以由开发人员扩展。因此，为了充分利用 MapReduce 的处理能力，了解 MapReduce 的基本组成细节是很有必要的。下面，我们会深入探讨 Map 阶段的细节内容，向这一目标努力。

关于 MapReduce 的详细介绍（包括各种算法的实现），有大量资料可供参考。一些不错的资源包括：《Hadoop 权威指南》《Hadoop 硬实践》以及《MapReduce 设计模式》。

1. Map阶段

接下来，我们会深入了解一个 MapReduce 任务在 Map 阶段涉及的主要组件。

`InputFormat` MapReduce 任务通过 `InputFormat` 类访问其数据。这个类实现了两个重要的方法。

- `getSplits()`

  该方法实现的是将输入分布到多个 Map 进程的具体逻辑。最常用的 `InputFormat` 是 `TextInputFormat`，它会为每个数据块生成一个输入分片，并将对应数据块的位置发送给 Map 任务。框架会针对每个分片启动一个 `Mapper` 执行处理。正因如此，开发人员常常假定 MapReduce 任务中，`Mapper` 的个数就是待处理数据集的数据块个数。

  这一方法决定了 Map 进程的个数，以及 Map 进程执行涉及的节点。不过，MapReduce 任务的开发人员可以重写这一方法，从而对要读取的数据获得完全的控制。举例来说，HBase 代码中的 `NMapInputFormat` 就允许用户直接设置执行任务的 `Mapper` 个数。

- `getReader()`

  这一方法为 Map 任务提供了 Reader 机制，赋予了 Map 访问待处理数据的能力。因为开发人员可以重写这一方法，所以 MapReduce 能够支持任意数据格式。只要能够提供一个将读入的数据转化成 Writable 对象的方法，就可以使用 MapReduce 框架处理该类数据。

`RecordReader` `RecordReader` 类读取数据块的内容，并将键值对的记录返回到 Map 任务。大多数的 `RecordReader` 实现起来极为简单：初始化 `RecordReader` 实例，使用的是需要读取的数据块在文件中的起始位置，以及该文件在 HDFS 中的 URI 地址。寻址到这一起始位置

之后，每次调用 `nextKeyValue()` 方法都会查找下一个行分隔符，并读取下一条记录。这种模式参见图 3-3。

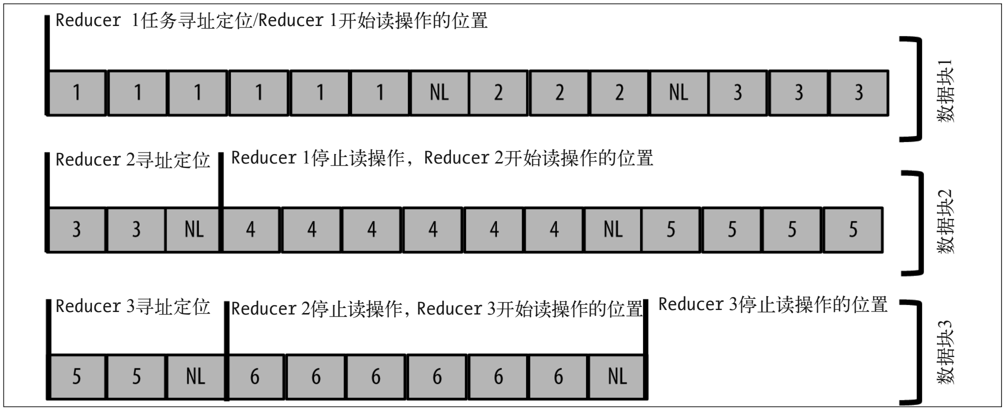


图 3-3：MapReduce 中的 `RecordReader`

MapReduce 框架以及这个生态系统中的其他项目提供了许多文件格式的 `RecordReader` 实现，包括带分隔符的纯文本、SequenceFile、Avro、Parquet，甚至还有不会读取任何数据的 `RecordReader`——将键值对传递给 `Mapper` 后，`NMapInputFormat` 返回的是 `NullWritable`。这样做的目的是保证 `map()` 函数会被调用一次。

`Mapper` 的 `setup()` 在 Map 任务的 `map` 方法调用之前，`Mapper` 的 `setup()` 方法会首先被调用一次。这一方法可以使开发人员初始化 Map 进程中会用到的变量和文件句柄。`setup()` 最常用的作用是从配置对象中获取配置项的值。

Hadoop 中的所有组件都可以通过 `Configuration` 对象配置，该对象包含一些键值对，会在任务执行时传递给 Map 和 Reduce JVM。该对象的内容可以参见 job.xml。默认情况下，`Configuration` 对象包含有集群的每个 JVM 成功执行所需的信息，如 NameNode 的 URI 地址，以及协调任务的进程（如果 Hadoop 在 MapReduce v1 的框架上运行，该进程为 JobTracker；如果 MapReduce 在 YARN 之上运行，该进程为 Application Manager）。

在设置阶段，Map 和 Reduce 任务开始执行之前，可以向 `Configuration` 对象中添加新的值。任务开始执行之后，`Mapper` 和 `Reducer` 可以随时访问 `Configuration` 对象，以获取这些值。以下是一个 `setup()` 方法的简单示例，它获取了 `Configuration` 中的值，以填充成员变量：

```
public String foobar;
public final String FOO_BAR_CONF = "custom.foo.bar.conf";

@override
public void setup(Context context) throws IOException {
  foobar = context.getConfiguration().get(FOO_BAR_CONF);
}
```

注意，放入 `Configuration` 对象中的值，均可在 JobTracker（MapReduce v1）或 Application Manager（YARN）中读取。一般来讲，二者都会有一个网页界面，且它们的地址无安全性

控制，任何人均可访问。因此，我们不建议把敏感信息（如密码）传递到 Configuration 对象中。一个推荐的做法是传递密码文件在 HDFS 上的 URI 地址，这样就可以拥有合适的访问权限控制。执行 MapReduce 的用户拥有有效的权限时，Map 和 Reduce 任务自然就可以读取该文件并获得相应的密码。

Mapper 的 map() Mapper 的核心就是 map() 方法。即使不需定制，直接使用 Map 任务中的默认组件，你仍然需要实现一个 map() 方法。该方法拥有三种输入：键（key）、值（value）及上下文（context）。其中，key 和 value 均可以通过 RecordReader 获得，它们包含 map() 方法需要处理的数据。context 对象为 Mapper 更多行为的实现提供了支持：将输出发送给 Reducer，从 Configuration 对象中读取值，计数器自增以汇报 Map 任务进度。

当 Map 任务输出数据，交由 Reducer 处理时，输出的数据会首先缓存和排序。MapReduce 试图在内存中对其进行排序，内存大小由 io.sort.mb 的配置参数确定。如果内存中的缓冲区过小，输出的数据将无法在内存中进行排序操作。此时，数据会吐到 Map 任务所在节点的本地磁盘上，在磁盘上排序。

Partitioner Partitioner 实现了在 Reducer 之间进行数据分区的逻辑。默认的 Partitioner 处理逻辑会获取键，得出标准的哈希函数散列，除以 Reducer 数。余数则决定这条记录的目标 Reducer。这样可以保证数据在 Reducer 之间是平均分布的，进而确保 Reducer 能够在相同的时间启动，而且在几乎同一时间执行结束。不过，如果 Reducer 处理的过程需要保证特定的值一起处理，那么你需要重写默认值，实现一个自定义的 Partitioner。

二次排序就属于这种情况。假设有一个时间序列，比如股票市场的价格信息，你可能希望让每个 Reducer 处理指定股票的所有交易，并按照交易的时间顺序排序，以了解股价与时间的相关性。在这个场景下，可以把股票编码与时间的组合（ticker-time）定义为 key。使用默认的 Partitioner 就会将同一个股票的多条记录发送到不同的 Reducer，所以此时就需要实现一个自定义的 Partitioner，以确保只按照股票代码（不适用时间戳）对记录进行分区，进而发送给 Reducer。

下面是这种 Partitioner 实现的简单示例：

```
public static class CustomPartitioner extends Partitioner<Text, Text> {
    @Override
    public int getPartition(Text key, Text value, int numPartitions) {
      String ticker = key.toString().substring(5);
      return ticker.hashCode() % numPartitions;
    }
}
```

我们可以从 key 中提取股票代码，并根据其值（而不是整个 key）的散列形成分区。

Mapper 的 cleanup() cleanup() 方法在 map() 方法处理了所有的记录之后调用。这里通常要执行文件的关闭操作，以及最后的报告和总结，比如将最终状态写入日志中。

Combiner MapReduce 中的 Combiner 可以提供一种简单的方法，减少 Mapper 与 Reducer 间的数据传输。让我们回顾一下单词统计的经典例子。在这个例子中，Mapper 处理每一行的输入，并拆分成单独的词，然后在输出的每个词后面加上“1”（表示当前的计数），如下所示。

```
the => 1
cat => 1
and => 1
the => 1
hat => 1
```

如果定义了 combine() 方法，你就可以对 Mapper 产生的值进行聚合操作了。这一方法会在 Mapper 执行的节点上执行，所以这种聚合行为减少了之后通过网络发送给 Reducer 的输出。Reducer 仍然需要把不同 Mapper 的结果聚合到一起，不过这些数据会明显减少。有一点是非常重要的，对于 Combiner 是否会执行到，我们是无法控制的。因此，Combiner 的输出应当与 Mapper 的输出格式相同，因为无论哪一种输出，Reducer 都会对其处理。另外，要注意 Mapper 的输出是有序的，因此可以假定 Combiner 的输入也是有序的。

在我们的例子中，Combiner 的输出如下。

```
and => 1
cat => 1
hat => 1
the => 2
```

2. Reducer

Reduce 任务没有 Map 任务那么复杂，不过还是有几个需要注意的组件。

Shuffle 在 Reduce 阶段正式开始之前，Reduce 任务会将 Mapper 的输出从 Map 节点复制到 Reduce 节点。每一个 Reducer 都需要从多个 Mapper 聚合数据，所以我们要让每一个 Reducer 按照 Map 任务的方式读取本地的数据。经由网络的数据复制是不可或缺的，因此集群内高吞吐的网络可以显著提高处理性能。这也是 Combiner 高效可用的主要原因。在通过网络发送出去之前，聚合 Mapper 的结果可以显著地加速这一阶段。

Reducer 的 setup() Reducer 的 setup() 步骤与 map 的 setup() 非常相似。这一方法在 Reducer 开始处理单个记录之前执行，通常用来初始化变量和文件句柄。

Reducer 的 reduce() 与 Mapper 的 map() 函数相似，reduce() 方法就是 Reducer 进行绝大多数数据处理的地方。然而在输入方面，Reducer 的 reduce() 有如下几个明显的不同点。

- key 是有序的。
- 参数从一个 value 变成了多个 values（从一次处理一个值变为处理多个值）。对于一个 key 来讲，输入会包含这个 key 对应的所有值，允许据此进行任何形式的聚合，也可以处理当前 key 的所有值。有一点非常重要，那就是一个 key 及其对应的值不会分散到多个 Reducer 上。这显而易见，不过开发人员经常会因为发现某个 Reducer 比其他的耗时久而感到吃惊。通常来讲，这是因为与其他 Reducer 相比，某个 key 的 Reducer 明显有多得多的值需要处理。这种分区后的数据倾斜是引发性能问题极为常见的原因。有经验的 MapReduce 开发人员在保证聚合结果正确的同时，会投入大量的精力确保数据分区后在 Reducer 上均衡。
- 在 map() 方法中，调用 context.write(K, V) 方法可以将输出放入缓存区，缓存区中的数据稍后会被排序，然后被 Reducer 读取。在 reduce() 方法中，调用 context.write(Km, V) 会将输出发送给 outputFileFormat，稍后我们会讨论后者。

Reducer 的 cleanup() 与 Mapper 的 cleanup() 方法类似。Reducer 的 cleanup() 方法在所有记录处理完毕之后才会调用。在这里可以关闭文件、将状态写入日志。

OutputFormat InputFormat 处理输入数据的读取，而 OutputFormat 负责数据的格式化和写入输出数据（通常是写入到 HDFS）。与输入格式相比，定制化的输出格式相对少一些。主要原因在于开发人员难以控制输入数据，因为输入数据来自遗留系统，以特定的格式存在。从另外一个角度来讲，输出数据是可以标准化的，有几种不错的格式可以用来作为输出格式。总会有人使用与众不同的新格式，但通常来讲，选择一种已有的输出格式就能满足需求。第 1 章讨论了大多数常见的数据格式，并说明了各自的适用场景。

OutputFormat 类的工作与 InputFormat 稍有不同。就大文件来讲，InputFormat 会将输出分片，以分配给多个 Map 任务，每个 Map 处理输入文件的一个子集。对于 OutputFormat 类来讲，一个 Reducer 只会写一个文件，因此在 HDFS 上，你可以看到一个 Reducer 对应一个输出文件。文件会以类似于 part-r-00000 的方式命名，文件名中的数字会随着 Reducer 任务数目的增加而递增。

有一个地方很有意思，需要注意：如果没有 Reducer，那么 Mapper 会调用 OutputFormat。这种情况下，输出文件会命名为 part-m-0000N，r 被替换成了 m。这只是命名的一般形式。不过，不同的 OutputFormat 命名的方式可能会不同。举例来说，Avro 的 OutputFormat 会使用 part-m-00000.avro 的命名形式。

3.1.2 MapReduce示例

在本章介绍的这些 Hadoop 数据处理方式里面，MapReduce 需要编写的代码最多。接下来的例子看起来会有些冗长，如果要在本章讲清 MapReduce 的一切，我们至少还需要 20 页的篇幅。在这里，我们来关注一个非常简单的示例：关联和过滤图 3-4 中的两个数据集。

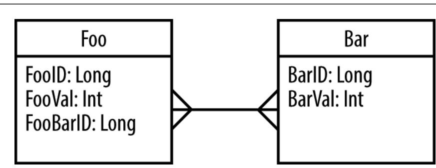


图 3-4　关联及过滤示例中的数据集

这个例子的数据处理要求包括以下几点。

- 通过 Foo 中的 FooBarId 与 Bar 中的 BarId，将 Foo 关联到 Bar 中。
- 过滤 Foo，移除 FooVal 大于用户指定的值（fooValueMaxFilter）的全部记录。
- 过滤关联产生的表，移除 FooVal 与 BarVal 的和大于另外一个用户指定参数（joinValueMaxFilter）的全部记录。
- 使用计数器（counter）跟踪移除的记录。

MapReduce 任务通常需要先创建一个 Job 实例，并执行它。如下是对应的代码示例。

```
public int run(String[] args) throws Exception {
  String inputFoo = args[0];
```

```
    String inputBar = args[1];
    String output = args[2];
    String fooValueMaxFilter = args[3];
    String joinValueMaxFilter = args[4];
    int numberOfReducers = Integer.parseInt(args[5]);

    Job job = Job.getInstance(); ❶

    job.setJarByClass(JoinFilterExampleMRJob.class);
    job.setJobName("JoinFilterExampleMRJob"); ❷

    Configuration config = job.getConfiguration();
    config.set(FOO_TABLE_CONF, inputFoo); ❸
    config.set(BAR_TABLE_CONF, inputBar);
    config.set(FOO_VAL_MAX_CONF, fooValueMaxFilter);
    config.set(JOIN_VAL_MAX_CONF, joinValueMaxFilter);

    job.setInputFormatClass(TextInputFormat.class);
    TextInputFormat.addInputPath(job, new Path(inputFoo)); ❹
    TextInputFormat.addInputPath(job, new Path(inputBar));
    job.setOutputFormatClass(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(output));

    job.setmapperClass(JoinFiltermapper.class); ❺
    job.setReducerClass(JoinFilterReducer.class);
    job.setPartitionerClass(JoinFilterPartitioner.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    job.setNumReduceTasks(numberOfReducers); ❻

    job.waitForCompletion(true); ❼
    return 0;
  }
```

现在来深入探讨任务配置的代码。

❶ 这是 `Job` 对象的构造函数，包含执行 MapReduce 任务所需的全部信息。

❷ 虽然这不是强制性的，但你最好给任务命名。这样可以保证任务在日志或 Web 界面上容易找到。

❸ 正如先前讨论的，配置任务时，可以创建一个 `Configuration` 对象，这样就可以在所有的 Map 和 Reduce 任务中利用它的值。在这里，我们将用于记录过滤的值放到 `Configuration` 中，任务执行之后，它们可以作为参数通过用户来定义，而不是硬编码到 Map 和 Reduce 任务中。

❹ 在这里我们设置输入和输出目录。这里的输入可以有多个，它们可以是文件，也可以是整个目录。除非使用了特定的 `OutputFormat`，否则输出路径一般只有一个，且必须是目录，这样每个 `Reducer` 就可以在这个目录下创建自己的输出文件。

❺ 代码的这个部分会配置任务中需要使用的类，包括 Mapper、Reducer、Partitioner 以及 InputFormat 和 OutputFormat。这个例子只需要定义 Mapper、Reducer 和 Partitioner。后面我们会介绍它们几个各自的实现代码。注意，OutputFormat 使用 Text 作为值的输出格式，使用 NullWritable 作为 key 的输出格式。这是因为我们只对最终输出的值感兴趣。key 的值将会被忽略，不会写入到 Reducer 的输出目录中。

❻ Mapper 的个数通过 InputFormat 控制，而我们需要直接指定 Reducer 的个数。如果设置 Reducer 的个数为 0，那么我们就会得到只含 Map 阶段的任务。默认的 Reducer 个数值在集群的层面定义，不过一般情况下，开发人员会重新配置这个值，因为他们更清楚具体任务对应的数据量和分区情况。

❼ 最后，我们将 MapReduce 任务提交给集群，等待运行的结果（成功或者失败）。

接下来，我们来关注 Mapper。

```
public class JoinFiltermapper extends
  mapper<LongWritable, Text, Text, Text> {

  boolean isFooBlock = false;
  int fooValFilter;

  public static final int FOO_ID_INX = 0;
  public static final int FOO_VALUE_INX = 1;
  public static final int FOO_BAR_ID_INX = 2;
  public static final int BAR_ID_INX = 0;
  public static final int BAR_VALUE_INX = 1;

  Text newKey = new Text();
  Text newValue = new Text();

  @Override
  public void setup(Context context) {

    Configuration config = context.getConfiguration(); ❶
    fooValFilter = config.getInt(JoinFilterExampleMRJob.FOO_VAL_MAX_CONF, -1);


    String fooRootPath =
      config.get(JoinFilterExampleMRJob.FOO_TABLE_CONF); ❷
    FileSplit split = (FileSplit) context.getInputSplit();
    if (split.getPath().toString().contains(fooRootPath)) {
      isFooBlock = true;
    }
  }

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String[] cells = StringUtils.split(value.toString(), "|");


    if (isFooBlock) { ❸
```

```
          int fooValue = Integer.parseInt(cells[FOO_VALUE_INX]);

          if (fooValue <= fooValFilter) { ❹
            newKey.set(cells[FOO_BAR_ID_INX] + "|"
                + JoinFilterExampleMRJob.FOO_SORT_FLAG);
            newValue.set(cells[FOO_ID_INX] + "|" + cells[FOO_VALUE_INX]);
            context.write(newKey, newValue);
          } else {
            context.getCounter("Custom", "FooValueFiltered").increment(1); ❺
          }
        } else {
          newKey.set(cells[BAR_ID_INX] + "|" +
                JoinFilterExampleMRJob.BAR_SORT_FLAG); ❻
          newValue.set(cells[BAR_VALUE_INX]);
          context.write(newKey, newValue);
        }
      }
    }
```

❶ 如前所述，`Mapper` 的 `setup()` 用于从 `Configuration` 对象中读取预先定义的值。在这段代码中，我们获取用于过滤的 `fooValMax` 变量的值，在后续的 `map()` 方法中会用到它。

❷ 每一个 Map 任务会读取对应数据块的数据，这里的数据块要么来自 Foo，要么来自 Bar 数据集。我们需要区分这两个数据，才能确保只过滤 Foo 表的数据，因此我们会将这一信息加入到输出的 `key` 中——`Reducer` 会使用输出的 `key` 关联操作数据集。在这部分代码中，`setup()` 方法识别出当前任务处理的数据块。在之后的 `map()` 方法中，我们会使用这个值来区分 `Foo` 和 `Bar` 数据集的处理逻辑。

❸ 在这里我们使用先前定义的数据块标识符。

❹ 此处使用 `fooValMax` 的值过滤。

❺ 这里需要最后指出一点，那就是在 MapReduce 中自增计数器的方法。计数器会有组名和计数器名，二者均可以通过 Map 和 Reduce 任务设置并执行自增操作。计数器会在任务执行结束时进行汇报，而且在任务的执行过程中，各种 UI 也会跟踪计数器的值。因此，使用计数器是向用户提供任务执行进度的好办法，同时还可以给开发人员提供一些信息，方便调试和排除故障。

❻ 注意我们对输出 `key` 的设置：首先是用来关联数据集的值，然后是一个“|”分隔符，接下来是记录标记（如果来自 `Bar` 数据集则标为 A，来自 `Foo` 则标为 B）。这就意味着，当 `Reducer` 获取到一个 `key` 和一组值并进行关联的时候，来自 `Bar` 数据集的值会先于来自 `Foo` 的值出现（`key` 是有序的）。要执行关联操作，我们只需将 `Bar` 数据集存放在内存中，直到 `Foo` 的值开始到达。如果没有记录标记将这些值排序，那么在关联的时候，就需要将整个数据集全部放入内存。

接下来我们集中讨论 `Partitioner`。这里需要实现一个自定义的 `Partitioner`，这是因为我们使用的 `key` 包含用于关联的 `key`，以及用于帮助排序的标记。我们要做的就是按照 ID 分区，从而保证用于关联的同一个 `key` 对应的两条记录能够出现在同一个 `Reducer` 中，不管它们之前来自 `Foo` 数据集还是 `Bar` 数据集。对于关联来说，这是必需的，因为单个 `Reducer`

需要关联同一个 key 的所有值。为此，我们只基于 ID 进行分区，整个组合 key 如下所示。

```
public class JoinFilterPartitioner extends Partitioner<Text, Text>{

  @Override
  public int getPartition(Text key, Text value, int numberOfReducers) {
    String keyStr = key.toString();

    String pk = keyStr.substring(0, keyStr.length() - 2);

    return Math.abs(pk.hashCode() % numberOfReducers);
  }

}
```

在 Partitioner 中，我们从 Map 输出的 key 中取出关联所用的 key，仅将 key 的这一部用于分区方法，如前所述。

接下来，让我们关注 Reducer，了解它是怎样关联两个数据集的。

```
public class JoinFilterReducer extends Reducer<Text, Text, NullWritable, Text> {

  int joinValFilter;
  String currentBarId = "";
  List<Integer> barBufferList = new ArrayList<Integer>();

  Text newValue = new Text();

  @Override
  public void setup(Context context) {
    Configuration config = context.getConfiguration();
    joinValFilter = config.getInt(JoinFilterExampleMRJob.JOIN_VAL_MAX_CONF, -1);
  }

  @Override
  public void reduce(Text key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {

    String keyString = key.toString();
    String barId = keyString.substring(0, keyString.length() - 2);
    String sortFlag = keyString.substring(keyString.length() - 1);

    if (!currentBarId.equals(barId)) {
      barBufferList.clear();
      currentBarId = barId;
    }

    if (sortFlag.equals(JoinFilterExampleMRJob.BAR_SORT_FLAG)) { ❶
      for (Text value : values) {
        barBufferList.add(Integer.parseInt(value.toString())); ❷
      }
    } else {
      if (barBufferList.size() > 0) {
        for (Text value : values) {
          for (Integer barValue : barBufferList) { ❸
```

```
            String[] fooCells = StringUtils.split(value.toString(), "|");

            int fooValue = Integer.parseInt(fooCells[1]);
            int sumValue = barValue + fooValue;

            if (sumValue < joinValFilter) {

              newValue.set(fooCells[0] + "|" + barId "|" + sumValue);
              context.write(NullWritable.get(), newValue);
            } else {
              context.getCounter("custom", "joinValueFiltered").increment(1);
            }
          }
        }
      } else {
        System.out.println("Matching with nothing");
      }
    }
  }
}
```

❶ 因为我们使用了一个标记辅助排序，所以首先需要根据给定的关联用 key 从 Bar 数据集中获取全部的记录。

❷ 在拿到需要的 Bar 数据集数据后，将其存储到内存中的链表里。

❸ 处理 Foo 数据集时，我们会通过循环的方式遍历 Bar 数据集，以执行关联操作。这就是一个*嵌套循环关联*（nested-loops join）。

3.1.3 MapReduce使用场景

正如上例所述，MapReduce 是一个较为底层的框架。开发人员需要时刻注意每个操作的细节，此处的代码涉及各种设置（setup）和引用（boilerplate）。因此，MapReduce 的代码通常会有很多 bug，维护成本较高。

不过，MapReduce 在本质上适合处理这些问题：文件压缩[1]、分布式数据复制、记录行级别的数据验证，等等。另外，在某些情况下，使用 MapReduce 编写代码可以更好地利用输入数据的属性，从而提高处理性能。举例来说，如果我们明确地知道输入数据是有序的，那么就可以在数据集合并时，利用 MapReduce 对其优化，而这是更高一级的抽象模型无法做到的。

只要能适应相应的编程模型，有经验的 Java 开发人员都应当使用 MapReduce。如果你处理的问题在本质上可以转化成 MapReduce 任务，或者在控制执行细节上有显著优点，则尤其如此。

注 1：我们会在第 4 章详细谈论压缩。

3.2 Spark

2009 年，加州大学伯克利分校（UC Berkeley）AMPLab 实验室的 Matei Zaharia 带领团队对 MapReduce 框架进行了改进研究。他们的结论是：虽然 MapReduce 模型非常适合海量数据处理，但是该框架受限于一个硬性的数据流模型，并不适用于其他应用。举例来说，在迭代式机器学习和交互式数据分析应用中，多次处理任务如果能够重复使用内存中缓存的数据集，将益处多多。MapReduce 将每次任务执行结束时的输出强制写入磁盘，并在下次任务执行时从磁盘中重新读入。另外，所有任务均要分成一个 Map 和一个 Reduce。如果框架能变得更为灵活，那么模型也会得到显著的改善和优化。

从这项研究中诞生了 Spark 项目，这个新的大数据处理模型改正了 MapReduce 的诸多缺点。随着这个项目逐渐为人所知，Spark 已经成为了包含 150 名代码贡献者的 Apache 的第二大顶级项目（HDFS 为第一大顶级项目）。

3.2.1 Spark概述

与 MapReduce 相比，Spark 的特别之处主要体现在以下几个方面。

DAG模型

回顾一下，MapReduce 模型只有两个处理阶段：Map 和 Reduce。使用 MapReduce 框架的时候，你必须通过串联 Map 和 Reduce 任务以及减少任务形成完整的应用。这种复杂的任务链就是所谓的有向无环图（Directed Acyclic Graphs，DAG），如图 3-5 所示。

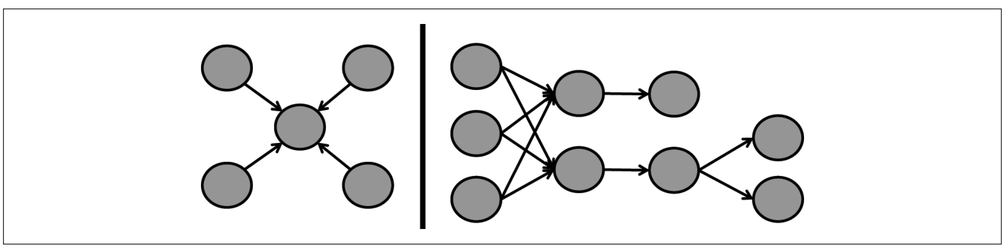


图 3-5：有向无环图

在工作流中，DAG 会包含互相连接的一系列操作。MapReduce 的 DAG 就是实现特定应用的一系列 Map 和 Reduce 任务。使用 DAG 来定义 Hadoop 应用并非首例，MapReduce 的开发人员也实现了 DAG，而且 DAG 还用到了基于 MapReduce 的更高一层的抽象接口上。Oozie 甚至支持用户以 XML 的形式定义 MapReduce 任务的工作流，并使用一个协调调度的框架监控任务的执行情况。

Spark 扩展的是引擎本身，它从应用逻辑的角度，增加了一些复杂的工具链，而不是对 DAG 抽象层进行外部扩展，形成模型。这样就可以让开发人员在一个任务之内编写复杂的算法和数据处理流水线，并允许框架从全局范围内优化该任务，从而提高整体性能。

想了解更多关于 Spark 的内容，参见 Apache Spark 的官网（http://spark.apache.org/）。现在暂时没有太多关于 Spark 的教材，不过由 Holden Karau 等人编写的《Spark 快速大数据分

析》[2]较全面地介绍了 Spark。关于 Spark 应用的更多高级用法，参见 Sandy Ryza 等人编写的《Spark 高级数据分析》[3]。

3.2.2 Spark组件概述

在介绍 Spark 的示例之前，我们首先从整体上了解一下 Spark 的各个组成部分。图 3-6 展示的就是 Spark 的主要组件。

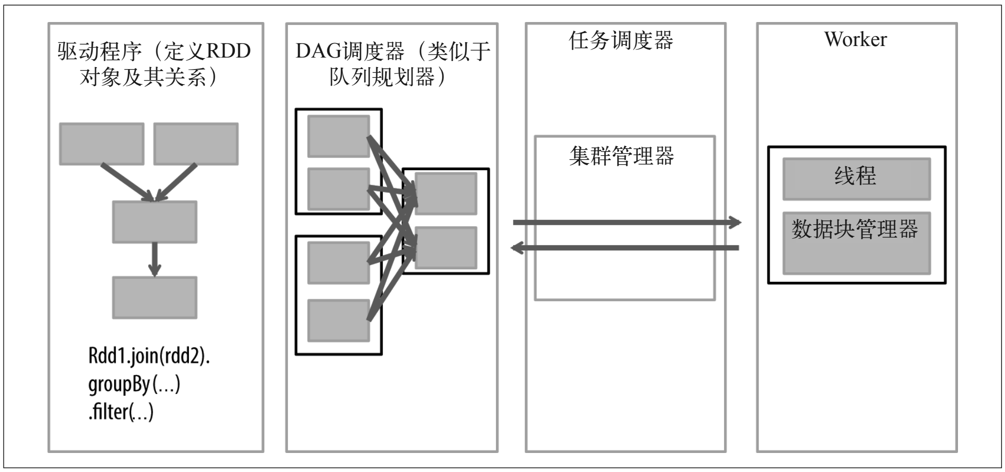


图 3-6：Spark 组件

让我们从左到右依次讨论图中的各个组件。

- 驱动程序指的是包含 main 函数的代码，它定义了*弹性分布式数据集*（Resilient Distributed Dataset，RDD）及其变换。RDD 是 Spark 编程中主要的数据结构，我们会在下一小节介绍相关知识。
- 基于 RDD 的并行操作会传递给 DAG 调度器（DAG scheduler），后者可以优化代码，并产生一个代表应用中数据处理步骤的高效 DAG。
- 最终的 DAG 会发送给*集群管理器*（cluster manager）。集群管理器知晓 Worker 的信息、已分配的线程、数据块的位置等，具有分配特定处理任务到 Worker 的功能。集群管理器还负责处理 Worker 失败时 DAG 的回放。正如之前所提到的，集群管理器可以是 YARN、Mesos 或者是 Spark 内置的集群管理器。
- Worker 接受和管理分配的任务和数据。Worker 执行自身的特定任务时，并不知晓整个 DAG，执行的结果会回传给驱动程序。

3.2.3 Spark基本概念

在探究过滤–关联–过滤（filter-join-filter）这一示例之前，让我们进一步讨论编写 Spark

注 2：此书已由人民邮电出版社出版。——编者注
注 3：此书已由人民邮电出版社出版。——编者注

应用时涉及的主要组件。

1. 弹性分布式数据集

RDD 是可序列化元素的集合，这一集合兴许可以分区。这个时候 RDD 就会存储到多个节点上。RDD 可能位于内存或磁盘。Spark 使用 RDD 减少磁盘 I/O，并维护内存中处理过的数据集，无需重启整个任务即可做到对节点失败容错。

通常情况下，RDD 由 Hadoop 的 InputFormat（比如，HDFS 上的一个文件）产生，或基于已存在的 RDD，经由变换操作得到。通过 `InputFormat` 创建一个 RDD 的时候，Spark 会根据 `InputFormat` 确定分区数目，这与 MapReduce 中的分片十分类似。如果 RDD 是由变换得到的，则有可能对数据进行 Shuffle 操作和重新分区，形成任意数目的分区。

RDD 存储它们的*血缘关系*，即到达当前状态经历的变换集合，始于创建 RDD 所用的第一个 `InputFormat`。如果数据丢失，Spark 可以根据血缘重演一系列的变换，重建丢失的 RDD，保证任务继续进行。

图 3-7 经常用于描述 Spark 中的 DAG。框图内部是 RDD 的分区，第二层是 RDD 以及单向的任务链。

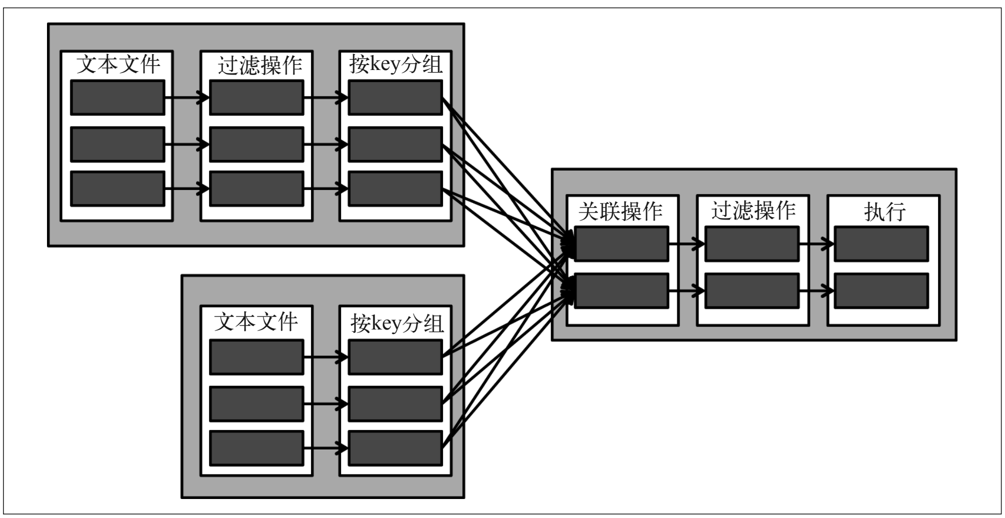


图 3-7 ：Spark DAG

现在，让我们假设黑色框图表示丢失的分区，如图 3-8 所示。Spark 会重新执行 Good Replay 框图对应的任务，以及 Lost Block 框图对应的任务，从而得到需要的数据，继续执行最终步骤。

注意，RDD 的类型有许多种，并不是每一个 RDD 上均可执行所有的变换。举例来说，不包含键值对的 RDD 就无法进行关联操作。

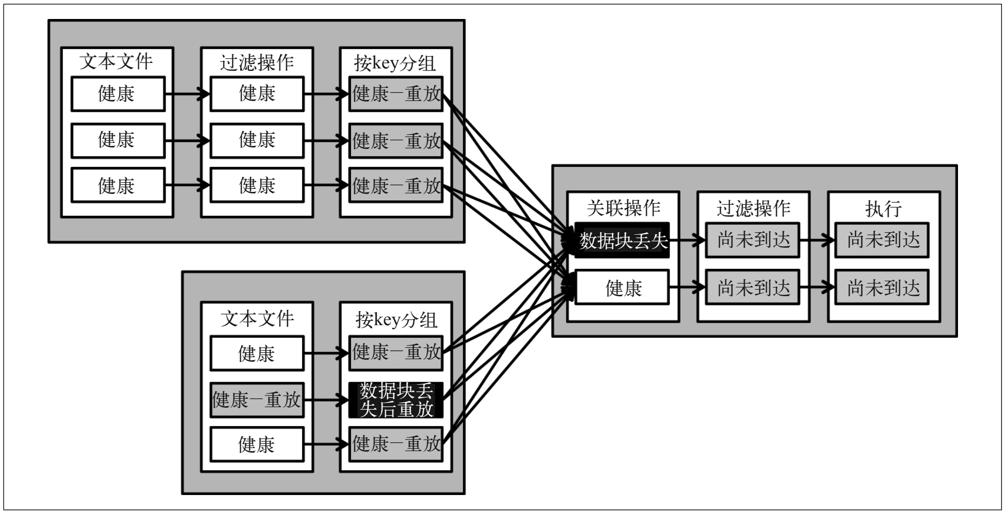


图 3-8 ：发生分区丢失后的 Spark DAG

2. 共享变量

Spark 有两种类型的变量用于执行节点之间的信息共享：*广播*（broadcast）变量与*累加器*（accumulator）变量。广播变量会发送到所有的远程执行节点，在这些节点上可以进行数据处理。它与 MapReduce 中 `Configuration` 对象扮演的角色类似。累加器变量也会发送到远端的执行节点，不过与广播变量不同的是，该类变量可以被执行器修改，且只能进行增加的操作。累加器与 MapReduce 中的计数器有几分相似。

3. `SparkContext`

`SparkContext` 对象代表到 Spark 集群的连接，可以用来创建 RDD、广播变量以及初始化累加器。

4. 变换

变换是以 RDD 作为输入，产生另外一个 RDD 的函数。RDD 是*不可变的*，因此变换不会修改其输入，而是返回一个更改后的 RDD。Spark 中的变换是*惰性的*，不会产生结果。调用变换函数只会创建一个带有某一变换（作为其血缘的一部分）的新 RDD。一系列变换只有在*调用操作*（action）时才会真正得到完整的执行。这样做能够提高 Spark 的性能，允许 Spark 更为智能地优化 DAG 图。这样做的缺点是让调试变得更为困难：异常在操作被调用时才会抛出，这与实际抛出异常的代码相隔甚远。需要谨记的是，处理异常的代码应当放置于操作附近（也就是异常抛出的地方），而不是变换的附近。

Spark 中有许多种变换。下面列出的一些变换可以让你了解变换函数究竟是什么样的。

- `map()`

  `map()` 对 RDD 的每一个元素应用一个方法，从而产生一个新的 RDD。这与对输入数据的每一个元素应用 MapReduce 的 `map()` 方法类似。比如：`lines.map(s=>s.length)` 表示输入一个由多个字符串构成的 RDD（即 `lines`），输出一个包含各个字符串长度的 RDD。

- `filter()`

  `filter()` 以布尔函数作为参数，对 RDD 的每一个元素执行该函数，并返回一个新的 RDD，只包含让该函数返回 `true` 的元素。比如：`lines.filter(s=>(s.length>50))` 返回的 RDD 都有包含 50 多个字符的行。

- `keyBy()`

  `keyBy()` 的输入是 RDD 的每个元素，输出是新 RDD 中的键值对。比如：`lines.keyBy(s=>s.length)` 返回以每行长度为键、每行内容为值的键值对的 RDD。

- `join()`

  `join()` 将两个键值对类型的 RDD 按照键进行关联操作。例如，假定有两个 RDD：`lines` 和 `more_lines`。二者中的每一项均是一个键值对：以行的长度为键、行的内容为值。`lines.join(more_lines)` 返回每一行的长度，以及对应该长度的（由分别来自 lines RDD 和 more_lines RDD 的内容构成的）一对字符串。结果集中的元素形如 `<length, <line, more_line>>`。

- `groupByKey()`

  `groupByKey()` 对一个 RDD 按照键进行分组操作。比如：在 `lines.groupByKey()` 返回的 RDD 中，每个元素都是以行长为键，对应行长的一组行作为值。在第 8 章中，我们会使用 `groupByKey()` 来获取单个用户对应的页面浏览集合。

- `sort()`

  `sort()` 对 RDD 排序，返回一个有序的 RDD。

注意，变换包括的函数与在 MapReduce 的 Map 阶段可以进行的操作类似，不过有一些函数，比如 `groupByKey()`，属于 Reduce 阶段。

5. Action

Action 方法输入 RDD，执行计算并返回结果给驱动程序。正如之前提到的，变换是惰性的，不会在调用时立即执行。Action 触发变换的执行。计算的结果可以是集合、打印到屏幕的值、保存到文件中的值或其他类似的结果。不过，Action 不会返回 RDD。

3.2.4 Spark的优点

接下来，让我们列举说明使用 Spark 的优点。

1. 简单

与 MapReduce 相比，Spark 的 API 非常简单整洁。因此，在实现相同逻辑的时候，本节的例子会比 MapReduce 的代码短许多。不熟悉 Spark 的人也可以很容易地读懂这部分代码。API 的易用性非常好，几乎没有必要在 Spark 之上进行高级抽象，而在 MapReduce 之上有很多抽象，如 Hive 或 Pig 等。

2. 功能丰富

Spark 从一开始就有意打造一个可扩展的、通用的并行处理框架。它可以支持流式数据的处理框架（即所谓的 Spark Streaming），以及图处理引擎，即 GraphX。因为 Spark 具有的

强大灵活性，所以我们期望未来 Spark 会涌现出大量面向特定需求的专用软件库。

3. 降低磁盘I/O

MapReduce 会在 Map 阶段结束时将数据写到本地磁盘，在 Reduce 阶段结束时则写到 HDFS 上。也就是说，处理 1TB 的数据，需要写 4TB 数据到磁盘，并通过网络分发 2TB 数据。当 MapReduce 应用有多个任务串联在一起的时候，情况会更加糟糕。

Spark 的 RDD 可以存储到内存中，以多个步骤处理，不需要额外的 I/O 就可迭代。这是因为它并不包含特定的 Map 和 Reduce 阶段，通常情况下，数据只是在处理开始时从磁盘上读出，在有必要持久化时写入磁盘。

4. 存储

Spark 为 RDD 的存储方式赋予了充分的灵活性。RDD 可以存储到单个节点的内存中，可以存储在内存中并在多个节点中存有副本，也可以持久化到磁盘。记住，持久化由开发者控制。RDD 在经历多个变换步骤（相当于多个 Map 和 Reduce 阶段）的过程中，可以不写任何数据到磁盘上。

5. 多语言支持

Spark 是使用 Scala 语言开发的，Spark 的 API 支持 Java、Scala 和 Python。这就允许开发人员使用自己最为熟悉的语言进行 Spark 开发。Hadoop 开发人员通常使用 Java API，而数据科学家通常更喜欢使用 Python 实现（这样就可以配合 Python 强大的数值处理库使用 Spark 了）。

6. 独立于资源管理器

Spark 支持使用 YARN 和 Mesos 作为资源管理器，同时还支持独立管理（Standalone）资源的模式。如果未来会有资源管理器方面的升级和调整，Spark 也是可以兼容的。同时，这还意味着，如果开发人员偏爱某一种资源管理器，Spark 也不会强制他们更换。因为每个资源管理器有各自的优点和局限，所以这一点还是非常有用的。

7. 交互式shell（REPL）

Spark 任务可以以应用程序的方式部署，这类似于 MapReduce 任务的执行方式。Spark 还包含一个 shell（通常也称作 REPL：Read-Eval-Print-Loop）。REPL 允许开发人员快速进行交互式数据开发，而且更方便验证代码。与客户协同工作时使用 Spark shell，你可以一边思考问题，一边迅速检查数据，交互式地得到答案。shell 还可以用来对程序进行验证和排错。

3.2.5　Spark示例

我们先看一段示例代码，然后再进行讲解。该示例使用了 3.1.2 节中的数据集，并采用相同的方式处理。

```
var fooTable = sc.textFile("foo")   ❶

var barTable = sc.textFile("bar")

var fooSplit = fooTable.map(line => line.split("\\|"))  ❷
```

```
var fooFiltered = fooSplit.filter(cells => cells(1).toInt <= 500)   ❸

var fooKeyed = fooFiltered.keyBy(cells => cells(2))   ❹

var barSplit = barTable.map(line => line.split("\\|"))   ❺

var barKeyed = barSplit.keyBy(cells => cells(0))

var joinedValues = fooKeyed.join(barKeyed)   ❻

var joinedFiltered =   ❼
joinedValues.filter(joinedVal =>
  joinedVal._2._1(1).toInt + joinedVal._2._2(1).toInt <= 1000)

joinedFiltered.take(100)   ❽
```

以上代码功能如下。

❶ 加载 Foo 和 Bar 两个数据集，形成两个 RDD。

❷ 将 Foo 的每一行切分成多个单元的集合。

❸ 过滤切分好的 Foo 数据集，仅保存第二列小于 500 的元素。

❹ 使用 ID 列作为键，将结果转化成键值对。

❺ 按照切分 Foo 的方式切分 Bar，并以 ID 为键将其转化成键值对。

❻ 关联 Bar 和 Foo。

❼ 过滤关联产生的结果集。此处的 `filter()` 函数以 joinedVal 作为输入。这个 RDD 包含一对 `Foo` 和 `Bar` 行。我们遍历每一行，查看两个数据集第一列的和是否小于 1000。

❽ 展示结果集中的前 100 条记录。注意，这是本代码中唯一一个 `Action`，因此代码中定义的整个变换操作链只会在此处触发。

这段代码看起来已经很简洁了，不过还可以实现得更加紧凑些。

```
//简短版
var fooTable = sc.textFile("foo")
  .map(line => line.split("\\|"))
  .filter(cells => cells(1).toInt <= 500)
  .keyBy(cells => cells(2))

var barTable = sc.textFile("bar")
  .map(line => line.split("\\|"))
  .keyBy(cells => cells(0))

var joinedFiltered = fooTable.join(barTable)
  .filter(joinedVal =>
    joinedVal._2._1(1).toInt + joinedVal._2._2(1).toInt <= 1000)

joinedFiltered.take(100)
```

代码的执行计划如图 3-9 所示。

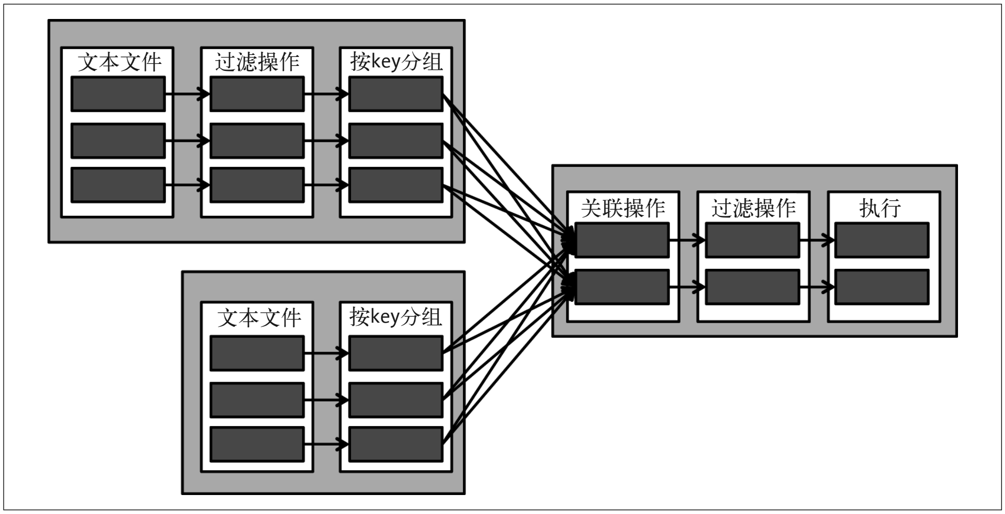
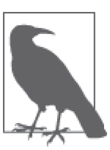


图 3-9：Spark 的执行

3.2.6 Spark使用场景

虽然现在 Spark 还是一个新生的框架，但它设计良好、速度飞快，具有易用性、扩展性，获得了高度的支持。我们有充足的理由相信它会快速地流行起来。Spark 仍然有些地方不够完善，但随着框架的逐步成熟，随着更多的开发者学会使用它，我们相信 Spark 的星星之火可以燎原，在大多数场景上使 MapReduce 黯然失色。Spark 十分擅长针对缓存在内存中的数据进行迭代操作，这使它成为了机器学习应用的不二之选。

除了可以取代 MapReduce，Spark 还提供了 SQL、图处理和流处理框架。这些项目相比 Spark 尚且稚嫩，它们能否流行开来，并在 Hadoop 生态系统中占据一席之地，还有待观察和确认。



另一个基于 DAG 的处理框架

将 Apache Tez（http://tez.apache.org）引入 Hadoop 生态系统是为了解决 MapReduce 中的限制。与 Spark 类似，Tez 也可以将数据处理表达成复杂的 DAG。Tez 架构设计的目标是提供优于 MapReduce 的性能和资源管理能力。对于原本以 MapReduce 作为执行引擎的任务（如 Hive、Pig），这种增强和改进使 Tez 很适合取代 MapReduce 成为新的执行引擎。

虽然 Tez 承诺能够极大地优化 Hadoop 上的数据处理，但它当前的状态更适合作为一个底层工具，支撑 Hadoop 上的执行引擎，而不是更高一级的开发工具。因此，本章不会详细地介绍 Tez，不过随着 Tez 日渐成熟，我们期待看到它成为更强大的 Hadoop 应用开发工具，同其他工具展开竞争。

3.3 抽象层

为了让 MapReduce 更好用，有许多项目开发出来。这些项目通过抽象隐藏了 MapReduce 的复杂性，让海量数据集的处理变得较为容易。Apache Pig、Apache Crunch、Cascading 以及 Apache Hive 均属于这种情况。这些抽象对应两类不同的编程模型：ETL（Extract, Transform, Load，抽取、转换、加载）和查询。我们首先讨论 ETL 模型，包括 Pig、Crunch 和 Cascading。接下来，我们会学习 Hive 及查询模型。

那么 ETL 模型与查询模型的不同点是什么？ ETL 过程以指定数据集作为输入，对其执行一系列的操作产生一个输出数据集，ETL 又对这一过程进行优化。而查询模型则向数据提出问题，获取答案。不过这不是分类的硬性标准，不能简单地一刀切。Hive 同时也是执行 ETL 操作的常用工具，第 10 章会进一步讨论这一点。但是，在一些场景中，与 Hive 这样的查询引擎相比，Pig 这样的工具能够在实现 ETL 工作流方面提供更好的灵活性。

虽然这些工具最初都是在 MapReduce 上作为抽象实现的，不过它们大多数都可以在其他的执行引擎上运行。举例来说，Tez 就可以作为引擎执行 Hive 查询。写作本书时，使用 Spark 做 Pig 和 Hive 后端的工作也在活跃地进行中。

在深入了解这些抽象之前，让我们先讨论它们的不同点。

Apache Pig

- Pig 提供了一种叫作 Pig Latin 的编程语言，能够更方便地在海量数据集上进行复杂的变换操作。
- Pig 提供一个用于快速开发的终端 Grunt shell，能够交互式地编写和运行脚本。
- Pig 提供 UDF 实现以支持自定义功能。
- 基于客户和用户的使用经验，Pig 是应用最为广泛的 Hadoop 数据处理工具。
- Pig 是顶级的 Apache 项目，有 200 多位代码提交人员。
- 正如先前提到的，尽管现在 Pig 最常用的执行引擎是 MapReduce，但 Pig 与代码无关。这就意味着，底层的执行引擎发生变化时无需修改 Pig 的代码。

Apache Crunch

- Crunch 应用程序是用 Java 语言编写的，因此熟悉 Java 的开发者不需要学习新的语言。
- Crunch 支持访问 MapReduce 的全部功能，在需要的情况下可以很容易地编写底层的代码。
- Crunch 支持与集成逻辑相隔离的业务逻辑，以进行代码隔离和单元测试。
- Crunch 是一个顶级 Apache 项目，大概有 13 位代码提交人员。
- 与 Pig 不同的是，Crunch 不是完全独立于执行引擎的，因为 Crunch 支持底层的 MapReduce 功能，这些特性在更换引擎时将不复存在。

Cascading

- 类似于 Crunch，Cascading 应用的代码也是用 Java 语言编写的。
- 类似于 Crunch，Cascading 也支持对 MapReduce 功能的全部访问，可以很容易地编写底层的代码。

- 类似于 Crunch，Cascading 支持与集成逻辑相隔离的业务逻辑，以进行代码隔离和单元测试。
- Cascading 项目大约有 7 位全职代码提交人员。
- 虽然 Cascading 不是一个 Apache 项目，但它持有 Apache 许可证。

接下来，让我们从 Pig 开始，深入了解这些抽象工具。

3.3.1 Pig

在 Hadoop 生态系统中，Pig 是基于 MapReduce 的最广泛使用的抽象工具之一。它是在 Yahoo 公司研发的，于 2007 年托管到了 Apache 基金会。

Pig 用户需要使用 Pig 专用的工作流语言 Pig Latin 编写查询语句，这种语句通过底层的执行引擎（如 MapReduce）进行编译并执行。Pig Latin 脚本首先会编译产生一个逻辑计划，然后生成物理计划。物理计划是可以通过底层引擎（如 MapReduce、Tez 或 Spark）执行的。指定查询对应的逻辑计划与物理计划是不同的，使用不同的执行引擎时，只有物理计划会发生变化。

想了解更多 Pig 知识，请浏览网站（http://pig.apache.org），或参阅《Pig 编程指南》。

3.3.2 Pig示例

下面的例子会让你看到，Pig Latin 是一种相当简单的自描述语言。浏览整个代码，你的思路在整体上与 Spark 类似。

```
fooOriginal = LOAD 'foo/foo.txt' USING PigStorage('|')
    AS (fooId:long, fooVal:int, barId:long);

fooFiltered = FILTER fooOriginal BY (fooVal <= 500);

barOriginal = LOAD 'bar/bar.txt' USING PigStorage('|')
    AS (barId:long, barVal:int);

joinedValues = JOIN fooFiltered by barId, barOriginal by barId;

joinedFiltered = FILTER joinedValues BY (fooVal + barVal <= 500);

STORE joinedFiltered INTO 'pig/out' USING PigStorage ('|');
```

对于这个脚本，你需要注意以下几点。

- *数据容器*

  这指的是 `fooOriginal` 和 `fooFiltered` 这种代表一个数据集的标识符。这在 Pig 中也称为*关系*，与 Spark 中的 RDD 概念类似（纵然它们在持久化语义上迥然不同）。提到 Pig 中的术语，一个关系就是一个*包*（bag），包也就是*元组*（tuple）的集合。元组是一系列值或者对象的集合。一个元组可以是包。一个包可以包含多个元组，一个元组可以包含多个包，以此类推。

• 变换函数

这里指的是类似于 FILTER 和 JOIN 这种可以变换关系的操作符。这里的逻辑也和 Spark 中的情况类似，这些变换不会立即执行。直到调用 STORE 命令时，变换才会真正执行，调用 saveToTextFile 时才会输出结果。

• 字段定义

这一点与 Spark 不同，字段及其数据类型是需要明确指出的（如 fooId:long）。这样可以使字段的数据处理更为简单，允许其他的外置工具从字段层次跟踪血缘关系。

• 不支持 Java

上面的例子不涉及导入某个类或对象的命令。在一定程度上，用户会受到 Pig Latin 的语言限制。从另外一个角度来讲，用户也避免了使用额外的编程语言接口进行数据处理的复杂性。如果的确需要使用编程接口，那么可以使用稍后提到的 Crunch 和 Cascading。

当执行类似于 Explain JoinedFiltered 这样的命令时，Pig 会展示该任务的查询计划。如下是来自 filter-join-filter 任务查询计划的输出。

```
#--------------------------------------------------
# MapReduce Plan
#--------------------------------------------------
MapReduce node scope-43
Map Plan
Union[tuple] - scope-44
|
|---joinedValues: Local Rearrange[tuple]{long}(false) - scope-27
|   |   |
|   |   Project[long][2] - scope-28
|   |
|   |---fooFiltered: Filter[bag] - scope-11
|       |   |
|       |   Less Than or Equal[boolean] - scope-14
|       |   |
|       |   |---Project[int][1] - scope-12
|       |   |
|       |   |---Constant(500) - scope-13
|       |
|       |---fooOriginal: New For Each(false,false,false)[bag] - scope-10
|       |   |
|       |   Cast[long] - scope-2
|       |   |
|       |   |---Project[bytearray][0] - scope-1
|       |   |
|       |   Cast[int] - scope-5
|       |   |
|       |   |---Project[bytearray][1] - scope-4
|       |   |
|       |   Cast[long] - scope-8
|       |   |
|       |   |---Project[bytearray][2] - scope-7
|       |
```

```
|       |---fooOriginal: Load(hdfs://localhost:8020/foo/foo.txt:\
PigStorage('|')) - scope-0
|
|---joinedValues: Local Rearrange[tuple]{long}(false) - scope-29
    |   |
    |   Project[long][0] - scope-30
    |
    |---barOriginal: New For Each(false,false)[bag] - scope-22
        |   |
        |   Cast[long] - scope-17
        |   |
        |   |---Project[bytearray][0] - scope-16
        |   |
        |   Cast[int] - scope-20
        |   |
        |   |---Project[bytearray][1] - scope-19
        |
        |---barOriginal: Load(hdfs://localhost:8020/bar/bar.txt:\
PigStorage('|')) - scope-15--------
Reduce Plan
joinedFiltered: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-40
|
|---joinedFiltered: Filter[bag] - scope-34
    |   |
    |   Less Than or Equal[boolean] - scope-39
    |   |
    |   |---Add[int] - scope-37
    |   |   |
    |   |   |---Project[int][1] - scope-35
    |   |   |
    |   |   |---Project[int][4] - scope-36
    |   |
    |   |---Constant(500) - scope-38
    |
    |---POJoinPackage(true,true)[tuple] - scope-45--------
Global sort: false
```

通过以上的查询计划你可以看到，这个 MapReduce 任务正如之前的 MapReduce 实现一样，而且 Pig 也在同一个地方执行了先前 MapReduce 代码中的过滤。因此，从内部实现来看，Pig 的代码与我们编写的 MapReduce 代码的执行速度是相同的。实际上，前者会更快一点。这是因为 Pig 会将执行过程中的数据存储成原始类型，而 MapReduce 则会翻来覆去地转换字符串。

3.3.3 Pig使用场景

这里总结一下关于 Pig 的讨论，如下是使用 Pig 执行 Hadoop 数据处理的理由。

- Pig Latin 易于阅读和理解。
- MapReduce 的复杂特性在 Pig 中化解了。
- 与相同功能的 MapReduce 任务相比，从代码行数及实现的角度看，Pig Latin 脚本非常简短。因此，Pig 脚本的维护成本比 MapReduce 任务低很多。

- 不需要对代码进行编译。Pig Latin 脚本可以在 Pig 控制台中直接运行。
- Pig 提供了强大的工具，可以查看任务是如何执行的。使用 `DESCRIBE joinedFiltered;` 可以查看集合中有哪些数据类型，使用 `Explain joinedFiltered;` 可以查看 Pig 用来获取结果的查询计划。

Pig 最大的缺点在于需要上手一门新的语言。对于许多开发者来讲，熟练掌握包、元组、关系等相关的概念是需要时间的。

通过 Pig 命令行访问 HDFS

Pig 还提供了一个独特的功能：访问 HDFS 的简单命令行接口。使用 Pig shell，你可以浏览 HDFS 文件系统，类似于在 Linux 上访问已挂载的 HDFS。如果手头的目录层次结构比较冗长，你希望在指定目录下执行文件系统命令（如 `rm`、`cp`、`pwd` 等），Pig shell 会很有用。通过在提示前面加上 `fs` 命令，你甚至可以用 `hdfs fs` 访问所有的命令，而且这些命令会在给定目录的上下文中执行。所以，即便不使用 Pig 开发应用级，也会觉得 Pig shell 访问 HDFS 是很有用的。

3.4 Crunch

Crunch 是基于 Google 的 FlumeJava[4] 设计开发的。它使数据流更加容易编进 Java，开发人员无需深入了解 MapReduce 的细节，不必考虑节点分布，也不必考虑 DAG 中的顶点。这里生成的代码与 Pig 比较类似，不过没有对字段的定义，因为 Crunch 会默认读取磁盘上的原始数据。

Spark 的核心是 `SparkContext`，而 Crunch 的核心是 `Pipeline` 对象（`MRPipeline` 或 `SparkPipeline`）。`Pipeline` 对象允许用户创建第一个 `PCollections`。Crunch 中的 `PCollections` 和 `PTable` 类似于 Spark 中的 RDD、Pig 中的关系。

调用 `done()` 方法会触发 Crunch 流水线的实际执行。Crunch 与 Pig 和 Spark 不同的一点在于，Pig 和 Spark 在执行 `Action` 时真正开始执行。而 Crunch 将执行延迟至调用 `done()` 方法。`Pipeline` 对象持有编译 Crunch 代码生成 MapReduce 或 Spark 工作流（以获取所需的结果集）的逻辑。

之前我们提到过，Crunch 支持 Spark 和 MapReduce，但 Crunch 代码不能 100% 兼容。这种限制主要集中在 `Pipeline` 对象上。对于不同的 `Pipeline` 实现，你可以获得不同的核心功能，如集合类型、函数类型等。这种不同在某种程度上是 Crunch 侧的需求冲突造成的。Crunch 希望提供 MapReduce 的底层功能，但是有一些功能在 Spark 中是不存在的。

想要了解更多关于 Crunch 的内容，参见 Apache Crunch 的主页（http://crunch.apache.org/）。另外，《Hadoop 硬实战》中提到了 Crunch 的一些内容，而《Hadoop 权威指南（第 4 版）》中有关于 Crunch 的章节。

注 4：注意，FlumeJava 仅在 Google 内部使用，是一个无法公开获取的项目。

3.4.1 Crunch示例

接下来的示例是一个执行 filter-join-filter 案例的 Crunch 程序。你会注意到，主要的逻辑代码在 `run()` 方法中。业务逻辑在 `run()` 方法之外进行了定义，会在流水线的不同地方触发。这与 Spark 中使用 Java 实现的方式类似，工作流程与业务逻辑就这样很好地分离开来。

```
public class JoinFilterExampleCrunch implements Tool {

  public static final int FOO_ID_INX = 0;
  public static final int FOO_VALUE_INX = 1;
  public static final int FOO_BAR_ID_INX = 2;

  public static final int BAR_ID_INX = 0;
  public static final int BAR_VALUE_INX = 1;

  public static void main(String[] args) throws Exception {
    ToolRunner.run(new Configuration(), new JoinFilterExampleCrunch(), args);
  }

  Configuration config;

  public Configuration getConf() {

    return config;
  }

  public void setConf(Configuration config) {
    this.config = config;

  }

  public int run(String[] args) throws Exception {

    String fooInputPath = args[0];
    String barInputPath = args[1];
    String outputPath = args[2];
    int fooValMax = Integer.parseInt(args[3]);
    int joinValMax = Integer.parseInt(args[4]);
    int numberOfReducers = Integer.parseInt(args[5]);

    Pipeline pipeline =
      new MRPipeline(JoinFilterExampleCrunch.class, getConf());  ❶

    PCollection<String> fooLines = pipeline.readTextFile(fooInputPath);   ❷
    PCollection<String> barLines = pipeline.readTextFile(barInputPath);

    PTable<Long, Pair<Long, Integer>> fooTable = fooLines.parallelDo(   ❸
        new FooIndicatorFn(),
        Avros.tableOf(Avros.longs(),
        Avros.pairs(Writables.longs(), Writables.ints())));

    fooTable = fooTable.filter(new FooFilter(fooValMax));   ❹

    PTable<Long, Integer> barTable = barLines.parallelDo(new BarIndicatorFn(),
```

```
      Avros.tableOf(Avros.longs(), Avros.ints()));

  DefaultJoinStrategy<Long, Pair<Long, Integer>, Integer> joinStrategy =   ❺
      new DefaultJoinStrategy
        <Long, Pair<Long, Integer>, Integer>
        (numberOfReducers);

  PTable<Long, Pair<Pair<Long, Integer>, Integer>> joinedTable =
    joinStrategy  ❻
      .join(fooTable, barTable, JoinType.INNER_JOIN);

  PTable<Long, Pair<Pair<Long, Integer>, Integer>> filteredTable =
      joinedTable.filter(new JoinFilter(joinValMax));

  filteredTable.write(At.textFile(outputPath), WriteMode.OVERWRITE);  ❼

  PipelineResult result = pipeline.done();

  return result.succeeded() ? 0 : 1;
}

public static class FooIndicatorFn extends
    MapFn<String, Pair<Long, Pair<Long, Integer>>> {

  private static final long serialVersionUID = 1L;

  @Override
  public Pair<Long, Pair<Long, Integer>> map(String input) {
    String[] cells = StringUtils.split(input.toString(), "|");

    Pair<Long, Integer> valuePair = new Pair<Long, Integer>(
        Long.parseLong(cells[FOO_ID_INX]),
        Integer.parseInt(cells[FOO_VALUE_INX]));

    return new Pair<Long, Pair<Long, Integer>>(
        Long.parseLong(cells[FOO_BAR_ID_INX]), valuePair);
  }
}

public static class FooIndicatorFn extends
    MapFn<String, Pair<Long, Pair<Long, Integer>>> {

  private static final long serialVersionUID = 1L;

  @Override
  public Pair<Long, Pair<Long, Integer>> map(String input) {
    String[] cells = StringUtils.split(input.toString(), "|");

    Pair<Long, Integer> valuePair = new Pair<Long, Integer>(
        Long.parseLong(cells[FOO_ID_INX]),
        Integer.parseInt(cells[FOO_VALUE_INX]));

    return new Pair<Long, Pair<Long, Integer>>(
        Long.parseLong(cells[FOO_BAR_ID_INX]), valuePair);
  }
```

```
}
public static class FooFilter extends
    Filterfn<Pair<Long,Pair<Long,Integer>>> {

  private static final long serialVersionUID =1L;

  int fooValMax;

  FooFilter(int fooValMax) {
    this.fooValMax = fooValMax;
  }

  @Override
  public boolean accept(Pair<Long, Pair<Long, Integer>> input) {
    return input.second().second() <= fooValMax;
  }
}

public static class FooFilter extends
    FilterFn<Pair<Long, Pair<Long, Integer>>> {

  private static final long serialVersionUID = 1L;

  int fooValMax;

  FooFilter(int fooValMax) {
    this.fooValMax = fooValMax;
  }

  @Override
  public boolean accept(Pair<Long, Pair<Long, Integer>> input) {
    return input.second().second() <= fooValMax;
  }
}

public static class BarIndicatorFn extends MapFn<String, Pair<Long, Integer>> {

  private static final long serialVersionUID = 1L;

  @Override
  public Pair<Long, Integer> map(String input){
    String[] cells = StringUtils.split(input.toString(), "|");

    return new Pair<Long, Integer>(Long.parseLong(cells[BAR_ID_INX]),
        Integer.parseInt(cells[BAR_VALUE_INX]));
  }
}

public static class JoinFilter extends
    FilterFn<Pai<Long, Pair<Pair<Long, Integer>, Integer>>> {

  private static final long serialVersionUID = 1L;
  int joinValMax;

  JoinFilter(int joinValMax) {
```

```
        this,joinValMax = joinValMax;
      }

      @Override
      public boolean accept(Pair<Long,
                      Pair<Pair<Long, Integer>,
                      Integer>> input) {

        return input.second().first().second() +
            input.second().second() <= joinValMax;
      }

    }
  }
```

下面我们对这段代码进行探究，了解一下究竟发生了什么。

❶ 首先我们通过 `Configuration` 对象创建了一个 `MRPipeline` 实例。注意，通过选用 `MRPipeline` 类，我们选择使用 MapReduce 作为底层的处理引擎。这里也可以换作使用 Spark 这样的执行引擎。

❷ 这里的两行代码将输入加入新创建的流水线中，并排序形成 `PCollection`。这样做与 Spark 使用 RDD 的思路一致。可以认为 `PCollection` 就是一个不可修改的分布式集合，在上面进行的绝大部分操作都可以分布化。类似于 Spark，`PCollection` 在物理上可能是不存在的，换句话说，它可能存储于内存中或磁盘上。

❸ 这里第一次调用 `parallelDo`，它类似于 Spark 中 RDD 的 `map()` 函数。该函数以分布式的方式遍历 `PCollection` 中的所有项目，要么返回另一个 `PCollection`，要么像本例中这样返回一个 `PTable`。注意，这里调用了一个名为 `FooIndicatorFn()` 的方法。后面有它的实现代码，现在我们只需知道以分布式的方式遍历每条记录时会调用到该方法。其他的参数则告诉 Crunch 我们接下来想要返回一个新的 `PTable`。`PTable` 与 `PCollection` 的不同点在于，前者包含我们需要的键值对，可以用来帮助关联。

❹ 通过一个分布式的过滤器创建一个新的 `PTable`。

❺ 定义接下来的关联使用的关联策略。在这里我们使用 DefaultJoinStrategy，不过还有其他的选项，如 BloomFilterJoinStrategy、MapsideJoinStrategy 以及 SharedJoinStrategy。在选择关联策略之前，需要了解数据集的特点。

❻ 这里是进行关联操作的示例代码。我们使用了新创建的关联策略，输入 `PTable` 以及内联或外联的定义等参数。

❼ 最后，我们将数据写入磁盘。注意这仍是一个分布式的写操作，因此你会看到输出目录中有多个文件，一个线程写一个文件。

3.4.2 Crunch使用场景

既然 Crunch 与 Spark 相似，是否有理由认为对于大多数的开发者来说，Crunch 会最终被 Spark 取代呢？考虑到几个因素，这也未必。使用基于 Spark 的 Crunch 可以获得抽象带来的好处，而且有抽象层的隔绝，底层执行引擎 Spark 可以替换。

然而，抽象层在带来好处的同时也需要付出代价。要做到充分地利用抽象，就需要理解底层的引擎，因此要学习 Spark 和 Crunch。另外，Spark 可能添加了新的功能，但这一点尚未同步到 Crunch 中，除非 Crunch 发生更新，否则将无法使用这一新功能。

大多数情况下，已经使用 Crunch 的开发人员会继续使用它。Crunch 与其他的抽象——如 Pig，以及不那么流行的 Cascading——存在竞争关系，正如 MapReduce 与底层执行引擎也存在着一定的竞争关系。如果你喜欢 Java，喜欢这种在底层执行引擎之上进行封装的抽象层思想，那么 Crunch 或许是一个不错的选择。

3.5 Cascading

在三个 ETL 抽象中，根据我们的经验，Cascading 是用户最少的一个。不过使用它的人觉得这个工具的确很有价值。看看代码，你会发现 Cascading 介于 Crunch 和 Pig 之间。它与 Crunch 类似的地方如下。

- 使用 Java 语言编码。
- Cascading 支持将业务逻辑与数据流剥离开来。
- Cascading 提供了访问底层接口的功能。

Cascading 与 Pig 在以下方面比较相似。

- Cascading 囊括了强类型字段的概念，允许列级别的血缘关系跟踪。
- 你可以通过 UDF 剥离业务逻辑，并实现自定义的功能。

想了解更多关于 Cascading 的信息，参见 Cascading 的主页（http://cascading.org/），或参考 Paco Nathan 所著的 *Enterprise Data Workflows with Cascading*。

3.5.1 Cascading示例

下面的代码使用 Cascading 实现了 filter-join-filter。第一次阅读这段代码，你可能会觉得它有些复杂，跟 Crunch 和 Spark 不大相同。不过仔细分析一下，你就会发现它们仍然有相近的地方。

```
public class JoinFilterExampleCascading {
  public static void main(String[] args) {
    String fooInputPath = args[0];
    String barInputPath = args[1];
    String outputPath = args[2];
    int fooValMax = Integer.parseInt(args[3]);
    int joinValMax = Integer.parseInt(args[4]);
    int numberOfReducers = Integer.parseInt(args[5]);

    Properties properties = new Properties();
    AppProps.setApplicationJarClass(properties,
        JoinFilterExampleCascading.class);
    properties.setProperty("mapred.reduce.tasks",
```

```
                        Integer.toString(numberOfReducers));
  properties.setProperty("mapreduce.job.reduces",
                        Integer.toString(numberOfReducers));

  SpillableProps props = SpillableProps.spillableProps()
      .setCompressSpill( true )
      .setMapSpillThreshold( 50 * 1000 );



  HadoopFlowConnector flowConnector = new HadoopFlowConnector(properties);  ❶

  //创建数据源和Sink对应的Tap
  Fields fooFields = new Fields("fooId", "fooVal", "foobarId");
  Tap fooTap = new Hfs(new TextDelimited(fooFields, "|"), fooInputPath);
  Fields barFields = new Fields("barId", "barVal");
  Tap barTap = new Hfs(new TextDelimited(barFields, "|"), barInputPath);  ❷

  Tap outputTap = new Hfs(new TextDelimited(false, "|"), outputPath);  ❸

  Fields joinFooFields = new Fields("foobarId");
  Fields joinBarFields = new Fields("barId");  ❹

  Pipe fooPipe = new Pipe("fooPipe");
  Pipe barPipe = new Pipe("barPipe");  ❺

  Pipe fooFiltered = new Each(fooPipe, fooFields, new FooFilter(fooValMax));  ❻

  Pipe joinedPipe = new HashJoin(fooFiltered, joinFooFields, barPipe,
      joinBarFields);  ❼
  props.setProperties( joinedPipe.getConfigDef(), Mode.REPLACE );


  Fields joinFields = new Fields("fooId", "fooVal", "foobarId", "barVal");
  Pipe joinedFilteredPipe = new Each(joinedPipe, joinFields,
      new JoinedFilter(joinValMax));

  FlowDef flowDef = FlowDef.flowDef().setName("wc")  ❽
      .addSource(fooPipe, fooTap).addSource(barPipe, barTap)
      .addTailSink(joinedFilteredPipe, outputTap);

  Flow wcFlow = flowConnector.connect(flowDef);  ❾
  wcFlow.complete();
}

public static class FooFilter extends BaseOperation implements Filter {

  int fooValMax;

  FooFilter(int fooValMax) {
    this.fooValMax = fooValMax;
  }

  @Override
```

```
    public boolean isRemove(FlowProcess flowProcess, FilterCall filterCall) {

      int fooValue = filterCall.getArguments().getTuple().getInteger(1);

      return fooValue <= fooValMax;
    }
  }

  public static class JoinedFilter extends BaseOperation implements Filter {

    int joinValMax;

    JoinedFilter(int joinValMax) {
      this.joinValMax = joinValMax;
    }

    @Override
    public boolean isRemove(FlowProcess flowProcess, FilterCall filterCall) {

      int fooValue = filterCall.getArguments().getTuple().getInteger(1);
      int barValue = filterCall.getArguments().getTuple().getInteger(3);

      return fooValue + barValue <= joinValMax;
    }
  }
}
```

让我们看看在 Cascading 中都发生了什么。

❶ Cascading 应用通常会拆分成四个阶段：配置阶段、集成阶段、处理阶段，和最后的调度阶段。在本例中，我们首先进行配置，并创建了 `HadoopFlowConnector`。

❷ 接下来，我们配置 `Tap`，即应用程序的输入数据。

❸ 在结束 Cascading 代码的集成部分之前，请注意我们定义输出的方式，这也是一个 `Tap` 对象。Cascading 包含 Source 和 Sink 两类 `Tap`，在这里我们创建了一个供输出使用的 Sink `Tap`。

❹ 从此处进入 Cascading 程序的数据处理步骤，在执行后续关联之前，定义关联使用的键。

❺ 在这里我们定义了 `Pipe` 对象。Crunch 使用 Java 编写分布式集合，与之不同的是，Cascading 要考虑的是 Pipe 和 `Tap`。这两个管道从数据源获取数据，执行关联后流向输出池。这里的管道在感觉和行为上与 Crunch 的 `PCollection`、Spark 的 RDD 比较相像。

❻ 使用 `Each()` 调用，将 `FooFilter()` 方法应用到 `fooPipe` 传递过来的每一个元组上。

❼ 这里是使用 Cascading 进行关联的第一个例子。与使用 Crunch 和 SQL（稍后会讨论）的情况类似，关联有很多选项和参数。

❽ 接下来，我们结束处理步骤，进入调度阶段。这里是 Cascading 中的“胶水”阶段。在这里我们将原始的 Source `Tap` 与 Sink `Tap` 放在了一起。

❾ 最后，在这里选择数据流和输出位置，开始执行任务。

3.5.2 Cascading使用场景

推荐使用 Cascading 的场景与适合使用 Crunch 的场景相同。选择通常取决于个人偏好、往日经验以及对特定编程模型的适应程度。毕竟，各种 ETL 抽象之间的区别比较小。

注意，以上讨论的抽象均属于 ETL 模型。在 3.6 节，重点会转向查询模型，开始讨论 Hive 和 Impala。

3.6 Hive

Hive 是第一个基于 MapReduce 创建的抽象引擎。Hive 诞生于 Facebook。数据分析师使用 Hive 分析 Hadoop 中的数据，而且能够使用熟悉的 SQL 语法，而不用学习如何编写 MapReduce 程序。

3.6.1 Hive概述

跟 Pig 相同，Hive 在 Hadoop 生态系统中已经存在了很长时间。Pig 要求学习新的语言，而 Hive 则允许你使用熟悉的语言抽象：SQL。自创建以来，由于支持 SQL，Hive 成为了 Hadoop 数据分析的一个热门选择。而且，Hive 现在甚至已经成为 Hadoop 上新型 SQL 实现——如 Impala、Presto（https://prestodb.io/）与 Spark SQL——的基础。

Hive 是一个成熟而且广为应用的项目，一直以来，它最大的缺陷就是性能不佳。不过，在评价了 Hive 及其性能之后，我们必须解释一下：Hive 正在往好的方向改变，而且这些改变能够帮助解决问题。

这里产生了性能问题，这很大程度上是因为 MapReduce 在 Hive 0.12 之前用作了 Hive 引擎。MapReduce 很实用，但是并不适合用于即席查询和交互式查询。不适用的原因很多，主要原因是 MapReduce 会频繁地读写磁盘，而且 MapReduce 任务的启动成本很高。因此，一个多表关联的查询会花费数分钟，不是因为数据量过大，只是因为读取和写入磁盘的操作太多。

Hive 社区很清楚这些性能问题，而且有许多人正在尝试解决它们。Hive 的改进方案如下。

- Hive-on-Tez
  该方案能够使用 `Tez`——一种比 MapReduce 性能更高的批处理引擎——作为 `Hive` 底层的执行引擎。Apache Hive 0.13.0 版本支持在 `Tez` 上运行 `Hive`。

- Hive-on-Spark
  Hive-on-Spark 与前面的 Hive-on-Tez 类似，但是它使用 Spark 作为 Hive 底层的执行引擎。该任务目前处于研发过程中，它的问题跟踪编号为 Hive-7292。

- 向量化执行查询
  该方案通过每次处理多行数据来减少 Hive 查询的 CPU 损耗，并减少处理数据时条件分支的数量。Apache Hive 0.13 是第一个支持向量化查询执行（vectorized query execution）的 Hive 版本，需要以特定的格式（如 ORC 与 Parquet）存储数据才能利用这一功能。

在 Hive 项目之外还有别的新项目出现，如 Impala、Spark SQL、Presto 与 Apache Drill。它们都能提供更快的 SQL-on-Hadoop。随后我们会在本章详细讨论相关内容。

前面提到的所有生态系统中的改进（无论是 Hive 项目之内的改进还是外部优化）都在使 SQL-on-Hadoop 的功能日益强大，速度日益提升。有一个要点需要注意：所有重新设计 Hive 项目的方法，以及新的项目都依然依赖于存储元数据的 Hive 元数据服务。因此，各种系统都能共享元数据，数据在各系统之间的传输或者多个系统之间的相互合作都变得更加容易。对于共享元数据，我们可以这样解释：比如，当用户在 Hive 中创建一个表或者对已经存在的表添加一个分区时，创建的表或添加的分区也可在 Impala 或其他系统中使用，反之亦然。这一点非常重要，而且对于开发者与 Hadoop 用户来说非常实在。Hive 元数据服务已经成为用户存储和管理元数据的标准。

图 3-10 为 Hive 高层架构图。Hive 包含一个名为 HiveServer2 的服务器，通过它可以使用 JDBC、开放式数据库连接（Open Database Connectivity，ODBC）及 Thrift 的客户端连接。HiveServer2 支持多种会话模式，均由 Hive Driver、编译器和执行器构成。HiveServer2 也与元数据存储服务通信，并使用关系型数据库存储元数据。正如第 1 章中所讲到的，元数据服务与对应的关系型数据库常统称为 Hive Metastore。

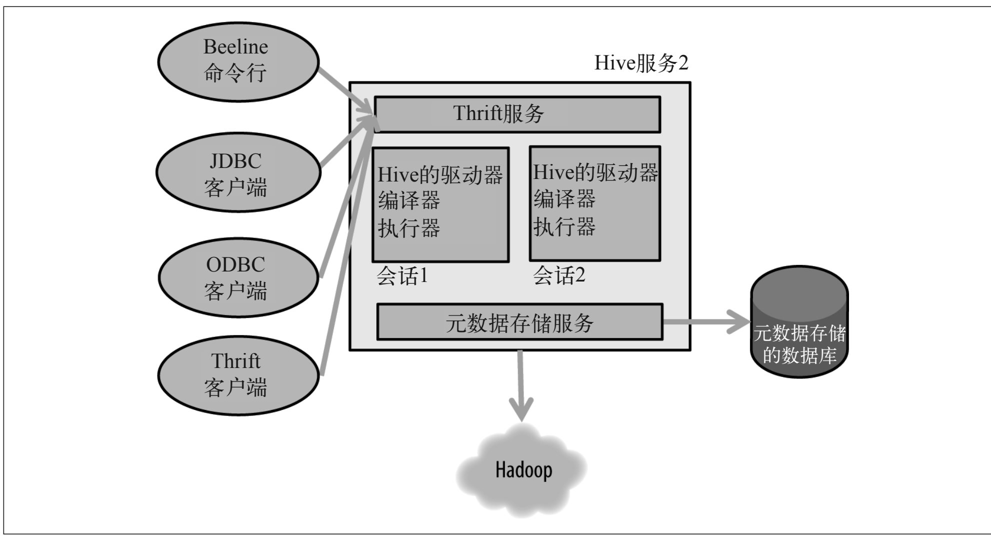


图 3-10：Hive 的架构

想要了解更多 Hive 相关内容，请参阅 Apache Hive 网站（http://hive.apache.org/）或者《Hive 编程指南》。

3.6.2 Hive示例

继续 filter-join-filter 的例子，我们来看一个使用 Hive 的实现。

首先需要为数据集创建表，如以下代码所示。我们将使用外部表，这样一来，删除该表只会删除元数据（Metastore 中的表名、列名、类型等信息）。HDFS 中的基础数据仍未动。

```
CREATE EXTERNAL TABLE foo(fooId BIGINT, fooVal INT, fooBarId BIGINT)
 ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
LOCATION 'foo';

CREATE EXTERNAL TABLE bar(barId BIGINT, barVal INT)
 ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
LOCATION 'bar';
```

你可能已经注意到了，上面的语法与传统 RDBMS 系统略有不同。不过，任何一个熟悉 SQL 的用户都应该能够理解这条查询语句的功能。简而言之，我们创建了两个表 foo 和 bar。表的数据存储格式为文本文件，用“|”作为列的分隔符，包含数据的文件位于 HDFS 中的 foo 与 bar 目录。

另外需要注意的是，在这些情况下我们也定义了存储格式。这里只是将数据存储为带分隔符的文本文件，但是在生产环境中，我们可能会使用一个更为优化的二进制格式（如 Parquet）存储数据。第 8 章将讨论具体的案例。这与大多数传统的数据存储（如 RDBMS）不同。在传统的数据存储中，数据被自动转化成适用于数据库的特定优化格式。

现在，完成了那些操作之后，我们可以运行命令来计算统计信息。这一步是可选的，不过一般不应跳过。这样一来，Hive 就可以基于数据分布，选择更适合数据的关联策略与执行计划。在 Hive 中计算示例中表的统计信息，命令如下所示。

```
ANALYZE TABLE foo COMPUTE STATISTICS;

ANALYZE TABLE bar COMPUTE STATISTICS;
```

一个名称为 hive.stats.autogather 的属性开关控制了 Hive 统计信息的自动生成，默认情况下设置为 true。但是，统计信息只有在通过 Hive 的 insert 语句插入数据时（如 INSERT OVERWRITE TABLE）才能自动计算。如果在 Hive 之外将数据载入 HDFS 中，或者使用类似于 Flume 的工具将数据输入 Hive 表的 HDFS 位置，那么用户需要运行一个 ANALYZE TABLE TABLE NAME COMPUTE STATISTICS 命令，明确地更新表的统计信息。

于是，这里运行只包含 Map 阶段的任务，读取数据、计算各种相关的统计信息（如 min、max 等）。这些表后续发生查询时，Hive 的查询计划器可以利用这些信息。

统计信息收集完毕之后，我们就可以继续了。下面是在 Hive 中执行一个 filter-join-filter 查询所需要的代码。

```
SELECT
  *
FROM
  foo f JOIN bar b ON (f.fooBarId = b.barId)
WHERE
  f.fooVal < 500 AND
  f.fooVal + b.barVal < 1000;
```

你可能注意到了，代码很简单，如果你熟悉 SQL 的话，则更是如此。因此，你似乎没必要

去学习 MapReduce、Pig、Crunch、Cascading 等工具了。

需要注意的是，就操作优化来说，Hive 自身做不到最好，有时甚至需要设定一些配置属性才能优化。比如，Hive 支持各种不同的分布式关联：map 关联（也称散列关联）、分桶关联（bucketed join）、排序后分桶合并关联（sorted bucketed merge join）与常规关联，等等。如果数据集符合特定的先决条件，那么与其他关联相比，某些关联策略可能会带来更好的性能。但是，因为较旧版本的 Hive 无法自动选择正确的关联策略，所以 Hive 编译器会依赖查询代码的提示来选择正确的关联策略。较新版本的 Hive 能够自动选择关联策略，而且随着项目的不断改进，Hive 也会自动完成越来越多的优化。

同样需要注意的是，尽管 SQL 有助于查询，但是它并不是表达所有数据处理格式的最佳语言。对于每一个用 SQL 表达的数据处理问题，你都需要看一下 SQL 是否适合。换句话说，要考虑这个问题会不会让 SQL 为难。使用简单过滤与聚合带来的问题适合 SQL。比如，如果想要知道上个月 Twitter 上最活跃的用户，用 SQL 查询会非常方便（假定你能够访问 Twitter 的数据集）。你只需要计算每一位用户的活动次数，然后得出活动次数最高的用户。另一方面，机器学习、文本处理与图算法通常不适合使用 SQL。如果需要根据兴趣和好友筛选向 Twitter 用户展示的广告，那么 SQL 不是恰当的工具。

与 Pig 相似，Hive 是一种 MapReduce 上的抽象工具（除非使用前面讲过的较新的执行引擎）。这表明 Hive 隐藏了后面的所有 MapReduce 任务。用户仍然应该养成习惯，查看 Hive 究竟在背后做了什么，以保证 Hive 按照用户的想法执行任务。用户只需在查询命令前面简单地增加一个词 `EXPLAIN`。下一个例子就会讲到 filter-join-filter 查询查询规划的内容。

就像你在查询规划中看到的那样，SQL 查询映射为 Hive 数据处理的 3 个阶段。注意本例中的第 3 个阶段，Hive 自动判断出要执行一个 map 关联（而不是性能较差的常规关联）。第 3 阶段根据 `SELECT * FROM foo f WHERE f.fooVal <500` 的结果生成一个哈希表。哈希表在关联时会存储到集群所有节点的内存中。然后，如第 4 阶段所示，数据从 `bar` 表读取，并与内存中包含 `foo` 表过滤值的哈希表进行关联操作。

```
EXPLAIN SELECT *
    > FROM foo f JOIN bar b ON (f.fooBarId = b.barId)
    > WHERE f.fooVal < 500 AND
    > f.fooVal + b.barVal < 1000
    > ;
OK
ABSTRACT SYNTAX TREE:


STAGE DEPENDENCIES:
  Stage-4 is a root stage
  Stage-3 depends on stages: Stage-4
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-4
  Map Reduce Local Work
    Alias -> Map Local Tables:
      f
```

```
        Fetch Operator
          limit: -1
      Alias -> Map Local Operator Tree:
        f
          TableScan
            alias: f
            Filter Operator
              predicate:
                  expr: (fooval < 500)
                  type: boolean
              HashTable Sink Operator
                condition expressions:
                  0 {fooid} {fooval} {foobarid}
                  1 {barid} {barval}
                handleSkewJoin: false
                keys:
                  0 [Column[foobarid]]
                  1 [Column[barid]]
                Position of Big Table: 1

  Stage: Stage-3
    Map Reduce
      Alias -> Map Operator Tree:
        b
          TableScan
            alias: b
            Map Join Operator
              condition map:
                   Inner Join 0 to 1
              condition expressions:
                0 {fooid} {fooval} {foobarid}
                1 {barid} {barval}
              handleSkewJoin: false
              keys:
                0 [Column[foobarid]]
                1 [Column[barid]]
              outputColumnNames: _col0, _col1, _col2, _col5, _col6
              Position of Big Table: 1
              Filter Operator
                predicate:
                    expr: ((_col1 < 500) and ((_col1 + _col6) < 1000))
                    type: boolean
                Select Operator
                  expressions:
                        expr: _col0
                        type: bigint
                        expr: _col1
                        type: int
                        expr: _col2
                        type: bigint
                        expr: _col5
                        type: bigint
                        expr: _col6
                        type: int
                  outputColumnNames: _col0, _col1, _col2, _col3, _col4
```

```
                    File Output Operator
                      compressed: false
                      GlobalTableId: 0
                      table:
                      input format: org.apache.Hadoop.mapred.TextInputFormat
                      output format:\
                      org.apache.Hadoop.Hive.ql.io.HiveIgnoreKeyTextOutputFormat
                      serde: org.apache.Hadoop.Hive.serde2.lazy.LazySimpleSerDe
        Local Work:
          Map Reduce Local Work

    Stage: Stage-0
      Fetch Operator
        limit: -1
```

3.6.3 Hive使用场景

尽管本章主要探讨使用 Hive 进行数据处理与查询的场景，但我们需要指出，Hive 的一部分（即 Hive 元数据存储）已经成为 Hadoop 生态系统中元数据存储的事实标准（表名、列名、类型等），详情参见 1.4。因此，无论使用哪种引擎，配置和使用 Hive 元数据存储来存储元数据都是最常见的选择。

当然，Hive 也适用于各种采用 SQL 表达的数据查询，尤其是容错要求高且长期运行的数据查询。除了数据查询，如果想要写入特点丰富、容错且批处理（也就是说，不是近实时的）的变换，或者在插件式的 SQL 引擎中执行 ETL 任务，Hive 也是不错的选择。让我们进一步讨论这些特点。

- SQL

  对于 Hadoop 来说，Hive 是一种 SQL 引擎。Impala 与其他一些未讨论的引擎（如 Apache Drill 与 Spark-SQL）也如此。所以，如果想以 SQL 的方式编写查询语句，那么你应该仅使用这种类别的引擎。

- 插件式

  Hive 可能是生态系统中插件式（pluggable）支持最好的 SQL 引擎。这里所说的插件式体现在很多方面。Hive 支持自定义数据格式、数据序列化和反序列化（Hive SerDes）。用户也能改变 Hive 底层的执行引擎，可以从 MapReduce 改变成 Tez，再变为 Spark（本书写作时仍在进行中）。所以，如果很需要支持底层的通用执行引擎之间的切换，那么 Hive 可能是一个不错的选择。

- 批处理

  Hive 是一种批处理引擎。这意味着你不能实时地获取结果。一般来说，其他引擎（如 Impala）的确比 Hive 速度快。因此，如果你对性能要求很高，那么 Hive 不是最好的选择。

- 容错

  Hive 是支持容错（fault-tolerant）的，而其他引擎（如 Impala）在本书写作时是不支持容错的。这意味着，如果处理部分数据查询的节点在运行时出现错误，那么 Impala 的整个查询将失败，但是在 Hive 中，底层的任务将会重试。如果数据查询长达数小时，

重试当然不可取，但是如果数据查询的时间较短，那么从客户端重新提交查询即可。因此，如果容错很重要，那么一般来说 Hive 是可以选用的。

- 功能强大

  如果 `Hive` 的速度很慢，我们为什么还要使用它呢？ `Hive` 是 `Hadoop` 上最成熟的 `SQL` 引擎。因此，在本书写作时，`Hive` 所包含的功能和特点比新的引擎要多很多。比如，在写作本书时，Impala 就不支持嵌套数据类型（`struct`、`array`、`map` 等）。如果支持嵌套格式很重要，那么 Hive 可能是一个不错的选择。

3.7 Impala

截止到 2012 年，Hadoop 已经在许多应用场景上大展身手，但是仍然只是作为一个低成本的存储和批处理 ETL 平台存在。恰逢其时，Google 公开了两篇描述低延迟查询引擎的论文：一篇涉及名为 F1 的支持容错的分布式 SQL 数据库，一篇涉及名为 Dremel 的可扩展的交互式即席查询引擎。在 2012 年，Impala 发布了。这是一个 Hadoop 之上的开源、低延迟 SQL 引擎，这一项目受到了 Google Dremel 论文的启发。

Impala 与当时 Hadoop 生态圈中其他的项目不同，它的基础不是 MapReduce。Impala 设计之初就优化了延迟，它的架构与传统的大规模并行处理（Massively Parallel Processing，MPP）数据仓库（如 Netezza、Greenplum、Teradata）相似。Impala 提供了近似于传统数据仓库的查询延迟和查询并发度，延迟比 MapReduce 上运行的 Hive 要低很多。

为了避免在 Hadoop 上形成数据孤岛，Impala 复用了 Hive 的 SQL 方言，并使用了 Hive 的元数据服务。如此，表只需定义一次，就可以在 Hive、Pig 和 Impala 中使用了。与 Hive 相似的是，Impala 支持将 HDFS 和 HBase 作为数据源，并支持大多数的常见数据格式（分隔符文本、SequenceFile、Avro 及 Parquet）。这一点使得 Impala 可以直接查询 Hadoop 上的所有数据，而不需要特殊的转换操作。

> 想要了解更多关于 Impala 的更多细节，请访问 Impala 网站（http://impala.io/），参考 John Russell 的 *Getting Started with Impala*。

另外，如本章先前提到的，还有其他一些开源项目支持 Hadoop 上的低延迟查询处理。包括 Presto 项目（http://prestodb.io/）和 Apache Drill 项目（http://drill.apache.org/）。

3.7.1 Impala概述

Impala 的架构是无共享的，这使得 Impala 支持系统级的容错和强大的可扩展性。用户和并发查询不断增加，Impala 仍然可以保持高性能。

Impala 从架构角度包含了 Impala 后台程序（impalad）、目录服务与 statestore。Impala 后台程序在集群的每一个节点上运行，每一个 impalad 都可以作为查询规划器、查询优化器及查询执行引擎。客户端连接 Impala，可以使用 JDBC、ODBC、impala-shell 或直接使用

Apache Thrift（http://thrift.apache.org/）连接到任意一个 impalad 上。所有的 impalad 都相同，可以互换。因此通常情况下客户端要连接到一个负载均衡器上，通过负载均衡器将连接转发到活动状态的 impalad。客户端连接的 impalad 会作为当前查询的查询规划器和查询协调器。

查询规划器的职责在于解析给定的 SQL 语句，并产生一个查询计划。查询协调器以查询计划作为输入，将计划的各个部分分配到各个 impalad 上执行。Impala 的架构如图 3-11 所示。

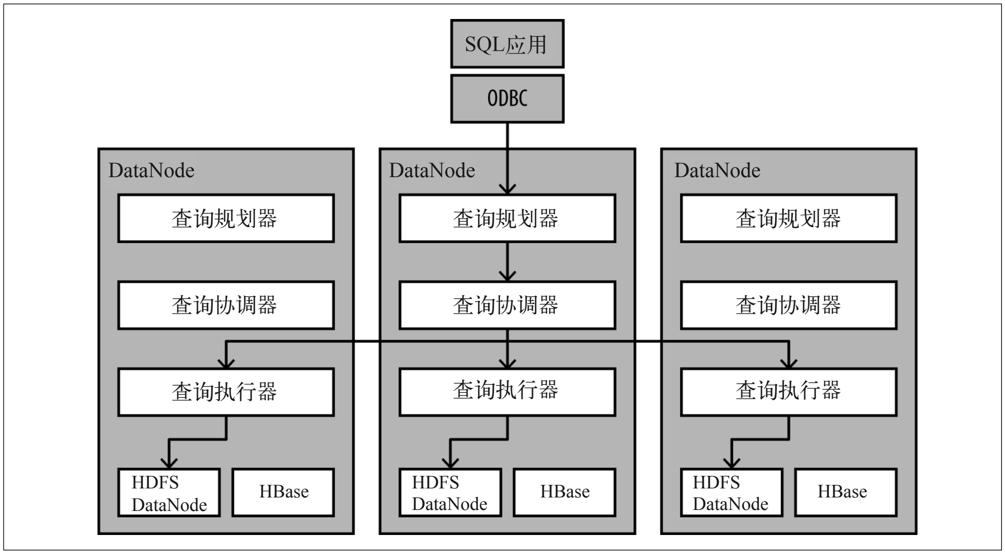


图 3-11 ：Impala 架构

注意，与其他数据库管理系统（如 RDBMS）不同的是，Impala 并没有实现底层的数据存储，因为它将数据存储到了 HDFS 和 HBase 中。Impala 也不是必须要实现数据表和数据库的管理方案，因为 Hive 元数据服务已经实现了这些工作。这就可以让 Impala 专注于它的核心功能，即尽可能高效地执行查询语句。

因为 Impala 要在 Hadoop 上做一个分布式的 MPP 数据库，所以需要实现 MPP 数据库中常见的各种分布式关联策略。本书写作时，Impala 实现了两种关联策略：*广播式散列关联*（broadcast hash join）及*分区后散列关联*（partitioned hash join）。如果读者希望了解这些关联策略的内部原理及在 Impala 中的实现方式，请参阅附录 A。

3.7.2 面向高速查询的设计

与 SQL-in-Hadoop 解决方案相比，Impala 有一些设计上的考虑能够降低查询延迟。

1. 高效利用内存

Impala 完全重写了查询引擎，不会受限于 MapReduce 引擎。初次扫描数据表时，数据会从磁盘上读取。接下来，随着多个处理阶段的进行，数据会保存在内存中。即使在不同的节点上进行了 Shuffle 操作，数据也会直接通过网络分发，分发之前不会先写到磁盘上。也就

是说，随着查询变得更加复杂，需要更多的处理阶段，Impala 性能的提升也会变得更为显著。相比之下，Hive 在每个阶段都要进行比较慢的磁盘数据读取，结束时还要写入磁盘。

这并非意味着 Impala 只能处理中间计算结果可以放入到内存中（聚合后）的查询。最初版本的 Impala 对较为消耗内存的查询存在这样的限制。这类的查询包括关联（要求过滤后的小表能够放入集群的内存中）、排序（单个节点在内存中执行部分排序操作）、分组和去重（每个去重后的键会存储到内存中以便聚合）。然而，在 Impala 2.0 及后续版本中，Impala 支持将超过内存的中间结果集吐到磁盘上。因此，使用新版本的 Impala，查询语句不会受限于这种对中间结果的硬性内存限制。Impala 仍会将数据放入内存，计算方式也和从前一样，但数据会在必要的时候吐到磁盘，然后再读取，尽管这样导致了较高的 I/O，降低了性能。

通常情况下，与基于 MapReduce 的处理相比，Impala 在提高性能时需要多得多的节点内存。这里推荐每个节点拥有最小 128~256GB 内存。Impala 会尽可能地多占用内存，而不是使用磁盘资源。这样做的一个缺点在于：当一个节点失联时，Impala 将无法恢复查询，而 MapReduce、Hive 可以。如果 Impala 查询处于执行过程中，而某个节点失联，那么这个查询就会失败。不过，在能够快速运行的查询场景上，如果查询失败只需重启整个查询，那么 Impala 仍然是可取的。大概花费几秒钟或五分钟左右的查询可以接受重启。不过，如果超过一个小时才能完成查询，那么使用 Hive 更为合适。

2. 长期运行的后台服务

与 Hive 使用的 MapReduce 引擎不同，Impala 后台服务是长期运行的进程。由于 Impala 一直处于运行状态，执行一个查询语句不会启动开销，也不会有 JAR 包的网络传输，或者类文件的加载。有人可能会问：是否可以在运行 MapReduce 任务的节点上运行 Impala 后台服务？还是需要在独立的节点上运行？我们推荐在所有的 DataNode 上运行 Impala，与 MapReduce 和其他处理引擎共存。这样就可以保证 Impala 能够通过本地节点而不是网络传输（即所谓的数据本地性）读取数据，这一点是减少延迟所必须的。Impala 与其他处理引擎间的资源竞争可以通过 YARN 动态管理，通过 Linux CGroups 静态分配。

3. 高效的执行引擎

Impala 是使用 C++ 语言实现的。这样做使 Impala 代码更为高效，允许单个 Impala 进程使用大量的内存，而不受 Java 垃圾回收机制（Garbage Collection，GC）的延迟影响。而且，Impala 还可以更好地利用向量化及特定 CPU 指令，以进行文本解析、CRC32 计算等，因为通过 JVM 是无法访问这些硬件特性的。

4. 对LLVM的使用

Impala 有一项重要的性能优化的技术：使用底层虚拟机（Low Level Virtual Machine，LLVM）编译查询，将查询中使用的所有方法编译成优化的机器码。这样做可以在多个方面加速 Impala 查询。首先，机器码中不包含多态（使用 Java 实现时需要对其处理），可以提高 CPU 对代码的执行性能。其次，产生的机器码使用了现代 CPU（如 Sandy Bridge 系列）提供的优化机制，能够提高 I/O 性能。第三，因为整个查询及其方法编译在同一个执行上下文中，所有的方法调用均是内联的，所以 Impala 不需切换上下文，在指令流水线上也没有分支，这会让整个执行过程更快。

Impala 的 LLVM 代码生成可以通过设置 disable_codegen 来关闭。这样做通常是为了解决问题，不过这样也可以准确地看到代码生成对查询有多大益处。

3.7.3 Impala示例

虽然 Impala 内部工作看起来相当复杂，但使用起来相当简单。启动 impala-shell（Impala 的命令行接口）就可以提交如下查询。

```
CONNECT <impalad主机名或负载均衡器的地址>;

--确保Impala从Hive元数据服务中获得了最新的元数据信息
INVALIDATE METADATA;

SELECT
  *
FROM
  foo f JOIN bar b ON (f.fooBarId = b.barId)
WHERE
  f.fooVal < 500 AND
  f.fooVal + b.barVal < 1000;
```

上面的代码连接到 Impala，通过 Hive 更新了元数据，并执行了我们的查询。跟本章前面的 MapReduce 代码相比，大多数的开发者更偏爱 SQL 版本，你可以从这里看出缘由。

想要在 Impala 中查看执行计划，在查询语句前面添加一个 EXPLAIN 即可。这一点与 Hive 相同，不过二者产生的查询计划完全不同。因为 Impala 实现了一个 MPP 架构的数据仓库，查询计划使用跟 Oracle 和 Netezza 类似的操作符。而 Hive 产生基于 MapReduce 的查询计划，二者有非常大的差异。

如下是查询对应的查询计划。

```
+------------------------------------------------------------------------------+
| Explain String                                                               |
+------------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=32.00MB VCores=2                     |
| WARNING: The following tables are missing relevant table                     |
| and/or column statistics.                                                    |
| default.bar, default.foo                                                     |
|                                                                              |
| 04:EXCHANGE [PARTITION=UNPARTITIONED]                                        |
| |                                                                            |
| 02:HASH JOIN [INNER JOIN, BROADCAST]                                         |
| | hash predicates: f.fooBarId = b.barId                                      |
| | other predicates: f.fooVal + b.barVal < 1000                               |
| |                                                                            |
| |--03:EXCHANGE [BROADCAST]                                                   |
| |  |                                                                         |
| | 01:SCAN HDFS [default.bar b]                                               |
| |    partitions=1/1 size=7.61KB                                              |
| |                                                                            |
| 00:SCAN HDFS [default.foo f]                                                 |
|    partitions=1/1 size=1.04KB                                                |
|    predicates: f.fooVal < 500                                                |
+------------------------------------------------------------------------------+
```

我们可以看到，Impala 首先会扫描数据表 `foo`，并使用查询语句中的谓词过滤。查询计划中包含了过滤谓词以及过滤后的数据表大小（这一数值是 Impala 优化器估算的）。过滤操作符的输出通过广播式关联的策略与表 `bar` 关联。查询计划还包括关联依据的列及另外一个过滤器。

Impala 还有一个 Web 界面，在每个 impalad 的 25 000 端口上运行。这一 Web 界面展示了查询的概述（Query Profile）。Query Profile 与查询计划类似，但是它在查询语句执行后生成。除了估算的大小之外，Query Profile 还包含其他的运行时信息，比如扫描数据表的速度、数据的实际大小、内存的使用量、执行时间，等等。这些信息对提高查询性能有很大的帮助。

3.7.4 Impala使用场景

正如我们之前讨论的，在 Hadoop 上执行 SQL，有若干工具可以选择。Hive 是其中使用最广泛的。那么什么时候使用 Hive，什么时候使用 Impala 呢？

我们认为问题的答案取决于具体的使用场景。更有意思的是，随着 Hive 和 Impala 的发展，答案也会发生改变。截至本书写作时，Impala 比 MapReduce 上运行的 Hive 要快许多，并支持高度的并发查询，但是 Impala 在容错方面不如 Hive，也不支持 Hive 的全部特性（如复杂数据类型 map、struct、array 等）。因此，对于要求高性能的场景，我们推荐使用 Impala。尤其是有上百个用户需要在 Hadoop 上并发地执行 SQL 查询的时候，Impala 可以很好地进行扩展，满足这一需求。不过，假如你的查询需要扫描非常多的数据（如上百 TB），那么使用 Impala 也需要特大量的 I/O，查询也需要几个小时。对于这样的场景，节点故障不可以强制要求重启恢复查询。我们推荐在这样的场景下使用 Hive。另外需要指出的是，在本书写作的时候，Impala 并不支持 Hive 上的某些特性。其中最重要的一点就是对嵌套数据类型处理的支持（这一点 Impala 已经在开发了）。因此，如果你的应用必须要有嵌套数据类型，那么我们还是推荐使用 Hive。随着 Hive 与 Impala 的发展，它们之间的差距会越来越小。

另外一个影响选择的重要因素是定制化文件格式的支持情况。Hive 支持这个生态系统中所有常用的文件格式（如带分隔符的文本、Parquet、Avro、RCFile、SequenceFile），也可以通过实现一个可插拔的 SerDe（Serializer/Deserializer）支持自定义的文件格式（如 JSON）。而 Impala 只支持 Hadoop 上的常用文件格式，不支持自定义的文件格式。在这种情况下，可以使用 Hive 直接读取特定格式的数据，或者使用 Hive 将数据转换成 Hadoop 生态系统中的常用文件格式，再使用 Impala 进行查询处理。别无他法。

另外还要注意，虽然在 Impala 与 Hive 中进行选择有些难度，在二者之间切换是非常简单的。因为二者都使用了相同的元数据服务。举例来说，Impala 可以读取 Hive 中创建的 Hive 表，反之亦然。你不需要在两个系统之间进行额外的转换操作。

3.8 小结

本章伊始就提到，我们的目标不是深入介绍 Hadoop 上现存的所有处理引擎，而是概述常用的数据处理候选工具，使你能够为特定的应用场景确定合适的工具。本章还要就你感兴

趣的工具提供恰当的参考资料。

第 4 章会使用本章提及的一些工具，介绍一些常见处理范式的实现。第 5 章会讨论特定的软件库和 API（如 GraphX 和 Giraph）如何支持 Hadoop 上的图处理需求。另外，我们还会谈到其他的软件库和 API 如何帮助实现 Hadoop 上的应用。我们没有那么多篇幅进行详细介绍，不过有些内容的确值得进一步探索和了解，如下所示。

- RHadoop

  在数据分析中，R 语言一直很流行，所以很多人对使用 R 语言处理大数据很感兴趣。R 语言自身的实现对处理海量数据产生了一些挑战，不过一些开源项目提供了处理 Hadoop 数据的 R 语言接口。这里面最为突出的就是 RHadoop。RHadoop 实际上是一些项目的集合，包括 rmr（提供了使用 R 语言实现 MapReduce 应用的接口）。

- Apache Mahout

  Mahout（http://mahout.apache.org/）是一个提供通用机器学习算法实现的软件库，包括推荐系统、分类、聚类等。虽然该项目的目标是对可扩展的机器学习算法提供可扩展的实现，不过 Mahout 中的实现并不都支持并行，也未必都设计得可以在 Hadoop 上运行。除了通用的机器学习算法，Mahout 项目中还包括处理其他任务的软件库。想要了解更多 Mahout 的相关知识，请参照其主页或《Mahout 实战》[5]。

- Oryx

  Oryx（https://github.com/OryxProject/oryx）是一个利用 Lambda（http://lambda-architecture.net/）架构辅助搭建机器学习应用的项目。它还处于项目初期，希望能够提供在 Hadoop 上部署分析模型的平台化支持。

- Python

  Python 是一门当之无愧的流行语言，对于数据科学家来说更是如此。如果选择了 Python，那么听到这个消息你一定很高兴：有许多框架都有意让 Python 在 Hadoop 上得到更为充分的利用。对于这些框架的介绍，请参见 Uri Laserson 所著的 *A Guide to Python Frameworks for Hadoop*。

注 5：此书已由人民邮电出版社出版。——编者注

第 4 章

Hadoop数据处理通用范式

了解了 Hadoop 上数据的访问方式和处理方式之后，我们接下来讨论如何使用第 3 章提到的工具解决一些常见的问题。本章会涵盖以下三种数据处理任务，它们也是 Hadoop 上数据处理常见的通用范式，在实现上具有一定的难度。

- 依主键（primary key）移除重复记录（合并去重）。
- 使用数据开窗分析。
- 更新时间序列数据。

接下来，我们会详细讨论以上范式的具体细节，并重点关注它们的实现，包括 Spark 和 SQL（对应 Impala 与 Hive）的使用。我们并未提供 MapReduce 版本的实现，这是因为 MapReduce 在这些逻辑的实现上较为复杂和冗长，而且人们也正向新型的处理框架（如 Spark）和数据抽象（如 SQL）靠拢。

4.1　模式一：依主键移除重复记录

Hadoop 上的数据经常存在重复的记录，原因如下。

- *数据采集阶段存在重复发送*

  正如本书中提到的那样，确保仅发送一次记录是一件不容易的事情，而且数据采集过程通常不会去处理重复的记录。

- *记录更新时的增量数据*

  HDFS 是一个“一次写入，多次读取”的文件系统。记录级的更改对于 HDFS 来讲并不容易。如果用例场景中有增量数据，那么在 Hadoop 上也会有对应主键（或复合主键）的数据集，更新后的记录会添加到这个数据集中。

本章会介绍如何处理第一类原因造成的完全重复记录。其他问题在别的章节探讨（比如，第 8 章的点击流案例研究将讨论第二类重复），记录的更新在示例中介绍。这种更新需要重写已有的数据集，以保证只展示记录的最新版本。

如果熟悉 HBase，你会发现这与 HBase 的工作方式类似。从上层看，HBase 的一个 Region 里有 HFile，其中包含一个键及其对应值。加入新数据时，这里会产生另外一个包含键和值的 HFile。执行合并（compaction）这一清理数据的过程时，HBase 会按键合并，删除重复数据，如图 4-1 所示。

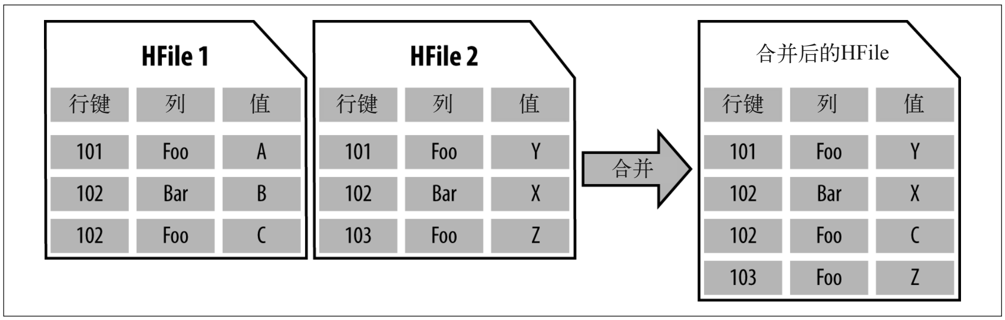


图 4-1 ：HBase 的合并示例

注意在以上示例中，我们忽略了其他一些复杂的内容，如 HBase 对数据多版本的支持等。

4.1.1　去重示例的测试数据生成

在讨论这一模式的实现示例之前，首先看一下生成测试数据的代码。我们将使用 Scala 对象 `GenDedupInput`，其功能是调用 HDFS 的 API，在 HDFS 上创建文件，并写入以下格式的记录。

```
{PrimaryKey},{timeStamp},{value}
```

我们将写入 x 条记录，主键有 y 种不同的取值。这就意味着，如果设定 x=100 和 y=10，那么我们需要给主键的每个独立取值生成大约 10 条重复的记录。

```
object GenDedupInput {
  def main(args:Array[String]): Unit = {
    if (args.length == 0) {
      println("{outputPath} {numberOfRecords} {numberOfUniqueRecords}")
      return
    }

    //定义存储数据的输出文件
    val outputPath = new Path(args(0))
    //Number of records to be written to the file
    val numberOfRecords = args(1).toInt
    //定义主键的不同取值个数
    val numberOfUniqueRecords = args(2).toInt

    //打开HDFS文件系统的访问句柄
```

```
    val fileSystem = FileSystem.get(new Configuration())

    //创建一个带缓冲区的writer
    val writer = new BufferedWriter(
      new OutputStreamWriter(fileSystem.create(outputPath)))

    val r = new Random()

    //以下循环完成全部记录的写操作
    //每个主键的不同取值对应若干条记录
    //对应条数由numberOfRecords/numberOfUniqueRecords确定
    for (i <- 0 until numberOfRecords) {
      val uniqueId = r.nextInt(numberOfUniqueRecords)
      //单条记录的格式为:{key}, {timeStamp}, {value}
      writer.write(uniqueId + "," + i + "," + r.nextInt(10000))
      writer.newLine()
    }

    writer.close()
  }
}
```

4.1.2 代码示例：使用Scala实现Spark去重

现在，我们完成了 HDFS 上测试数据的创建，接下来关注 `SparkDedupExecution` 对象中对记录去重的代码。

```
object SparkDedupExecution {
  def main(args:Array[String]): Unit = {
    if (args.length == 0) {
      println("{inputPath} {outputPath}")
      return
    }

    //设定传入参数
    val inputPath = args(0)
    val outputPath = args(1)

    //初始化Spark的SparkConf与SparkContext
    val sparkConf = new SparkConf().setAppName("SparkDedupExecution")
    sparkConf.set("spark.cleaner.ttl", "120000");
    val sc = new SparkContext(sparkConf)

    //读取HDFS上的数据
    val dedupOriginalDataRDD = sc.hadoopFile(inputPath,
      classOf[TextInputFormat],
      classOf[LongWritable],
      classOf[Text],
      1)

    //将数据解析成Key-Value的形式
    val keyValueRDD = dedupOriginalDataRDD.map(t => {
      val splits = t._2.toString.split(",")
      (splits(0), (splits(1), splits(2)))})
```

```
    //按照Key进行Reduce操作,从而保证每个主键取值只保留一条记录
    val reducedRDD =
        keyValueRDD.reduceByKey((a,b) => if (a._1.compareTo(b._1) > 0) a else b)

    //将数据格式化成可读格式,并写回到HDFS上
    reducedRDD
      .map(r => r._1 + "," + r._2._1 + "," + r._2._2)
      .saveAsTextFile(outputPath)
  }
}
```

我们将代码拆解开，进一步分析其功能。跳过获取用户参数、生成 SparkContext 的初始化代码，让我们关注从 HDFS 获取带有重复记录的数据的代码。

```
val dedupOriginalDataRDD = sc.hadoopFile(inputPath,
   classOf[TextInputFormat],
   classOf[LongWritable],
   classOf[Text],
   1)
```

在 Spark 中，数据有很多种读取方式。这一示例中使用 hadoopFile() 方法，以便描述如何使用输入格式。如果有 MapReduce 方面的编程经验，你就会比较熟悉 TextInputFormat，这是 Hadoop 上最为基础的输入格式。在 Spark 或 MapReduce 任务中，使用 TextInputFormat 可以将输入目录解析成目录下的文件集合，接下来才可以形成由不同子任务分别处理的数据块。

接下来的一段代码首先是一个 map() 函数：

```
val keyValueRDD = dedupOriginalDataRDD.map(t => {
  val splits = t._2.toString.split(",")
  (splits(0), (splits(1), splits(2)))})
```

以上代码将会在不同的 Worker 上并行执行，将输入的记录解析成由 key 和 value 组成的 Tuple 对象。

这种键值结构正是后续代码所需要的，后面会用到 reduceByKey() 方法。根据方法名的含义可以猜测到，使用 reduceByKey() 方法首先需要一个 key。

接下来，让我们关注调用 reduceByKey() 的代码片段。

```
val reducedRDD =
    keyValueRDD.reduceByKey((a,b) => if (a._1.compareTo(b._1) > 0) a else b)
```

reduceByKey() 方法的参数是一个函数，该函数输入一左一右两个值，输出一个相同类型的值。reduceByKey() 的目标是合并同一个 key 的所有 value。在单词统计的示例中，使用该函数可以累加每一个单词的个数，以获得总数。在本示例中，输出的 a 和 b 是字符串，返回的是二者中较大的一个。我们要根据主键的值进行 Reduce 操作，这个函数可以确保每一个主键只有一条输出记录。因此，这里基于主键选出的最大的键值进行数据去重。

最后一部分的代码将结果数据写回 HDFS。

```
reducedRDD
    .map(r => r._1 + "," + r._2._1 + "," + r._2._2)
    .saveAsTextFile(outputPath)
```

就这样，Spark 中的每个分区都得到了一个纯文本格式文件，这与 MapReduce 任务执行后期，为每一个 `Mapper` 或 `Reducer` 输出一个文件的机制相似。

4.1.3 代码示例：使用SQL实现去重

现在让我们使用 SQL 实现去重——更准确地说，是使用 HiveQL 实现去重。本章的示例在 Hive 和 Impala 中均适用。首先，我们用一条数据定义语言（Data Definition Language，DDL）语句将测试数据放到数据表中。

```
CREATE EXTERNAL TABLE COMPACTION_TABLE (
  PRIMARY_KEY STRING,
  TIME_STAMP BIGINT,
  EVENT_VALUE STRING
  )
 ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
 STORED AS TEXTFILE
 LOCATION 'compaction_data';
```

这样我们就有了一张表，接下来，让我们关注执行去重操作的语句。

```
SELECT
 A.PRIMARY_KEY,
 A.TIME_STAMP,
 MAX(A.EVENT_VALUE)
FROM COMPACTION_TABLE A JOIN (
  SELECT
   PRIMARY_KEY AS P_KEY,
   MAX(TIME_STAMP) as TIME_SP
  FROM COMPACTION_TABLE
  GROUP BY PRIMARY_KEY
  ) B
WHERE A.PRIMARY_KEY = B.P_KEY AND A.TIME_STAMP = B.TIME_SP
GROUP BY A.PRIMARY_KEY, A.TIME_STAMP
```

在这里，我们执行了一个两层的 SQL 查询。内层 `SELECT` 查询的作用是获取对应最新 `TIME_STAMP` 的所有不同的 `PRIMARY_KEY` 记录。外层的 `SELECT` 语句则根据内层 `SELECT` 语句查询的结果，取出对应的最新 `EVENT_VALUE` 值。另外，要注意的是，我们使用 `MAX()` 求取 `EVENT_VALUE` 的最大值，这是因为我们只需要保留单个值，如果同一个时间戳下有两个 `EVENT_VALUE`，那么我们就选择较大的那个作为去重后的记录。

4.2 模式二：数据开窗分析

开窗函数（windowing function）支持基于一定的窗口（例如特定的时间片），在有序的事件序列上进行扫描操作。这种处理范式功能强大且用途广泛。

- 在金融行业，开窗函数可以用来研究证券的价格变化。
- 在传感器监控领域，开窗函数可以基于异常值预测故障。
- 开窗函数也可用于客户流失分析，基于客户的行为模式预测客户是否流失。
- 在游戏领域，开窗函数可以用来识别用户从新手到专家的晋级趋势。

我们用金融领域的一个案例来深入介绍数据开窗：寻找股票价格的高峰与低谷，理解证券价格的变化。高峰记录是指价格高于其前后紧邻记录价格的记录。而低谷则与之相反，指的是两侧紧邻记录价格较高的记录，如图 4-2 所示。

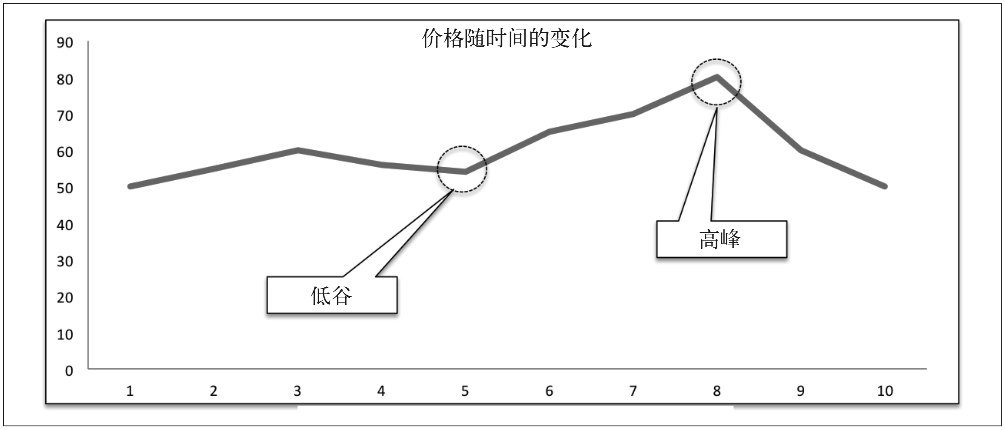


图 4-2：一段时间内股票价格的高峰与低谷

为了找出这个例子中的高峰和低谷，这里需要维护股票价格的一个窗口。

注意，这是一个简单的例子，SQL 和 Spark 都可以实现。随着开窗分析日渐复杂，SQL 会渐渐变得不太适用。

4.2.1 生成开窗分析的示例数据

接下来我们创建一个测试数据集，其包含值不断上升、下降，类似于股票价格。以下代码示例的输入参数与上一代数据生成工具（`numberOfRecords` 和 `numberOfUniqueIds`）相同。不过，这里产生的记录会有所不同。

- 主键

  主键是待分析事件序列的标识符。例如：股票代号（stock ticker symbol）就可以是一个主键。主键会基于 `numberOfUniqueIds` 输入参数。

- 增量计数器

  在生成的数据中，每一条记录这一字段的取值均是唯一的。

- 事件取值

  给定的主键会有一个随机记录集合，包含上升和下降的取值。

让我们看一下产生该测试数据的代码：

```
def main(args: Array[String]): Unit = {
  if (args.length == 0) {
   println("{outputPath} {numberOfRecords} {numberOfUniqueIds}")
   return
  }
```

```
    val outputPath = new Path(args(0))
    val numberOfRecords = args(1).toInt
    val numberOfUniqueIds = args(2).toInt

    val fileSystem = FileSystem.get(new Configuration())

    val writer =
      new BufferedWriter( new OutputStreamWriter(fileSystem.create(outputPath)))

    val r = new Random()

    var direction = 1
    var directionCounter = r.nextInt(numberOfUniqueIds * 10)
    var currentPointer = 0

    for (i <- 0 until numberOfRecords) {
      val uniqueId = r.nextInt(numberOfUniqueIds)

      currentPointer = currentPointer + direction
      directionCounter = directionCounter - 1
      if (directionCounter == 0) {
        var directionCounter = r.nextInt(numberOfUniqueIds * 10)
        direction = direction * -1
      }

      writer.write(uniqueId + "," + i + "," + currentPointer)
      writer.newLine()
    }

    writer.close()
  }
```

4.2.2 代码示例：使用Spark分析数据的高峰和低谷

现在，让我们来关注使用 Spark 实现这一范式的代码。下面的代码的确有点长。我们会在后面进行深入的探讨，帮助你理解它。

以下代码可以在 SparkPeaksAndValleysExecution.scala 文件中找到：

```
object SparkPeaksAndValleysExecution {
  def main(args: Array[String]): Unit = {
    if (args.length == 0) {
      println("{inputPath} {outputPath} {numberOfPartitions}")
      return
    }

    val inputPath = args(0)
    val outputPath = args(1)
    val numberOfPartitions = args(2).toInt

    val sparkConf = new SparkConf().setAppName("SparkTimeSeriesExecution")
    sparkConf.set("spark.cleaner.ttl", "120000");

    val sc = new SparkContext(sparkConf)
```

```
//数据读取 ❶
var originalDataRDD = sc.hadoopFile(inputPath,
  classOf[TextInputFormat],
  classOf[LongWritable],
  classOf[Text],
  1).map(r => {
  val splits = r._2.toString.split(",")
  (new DataKey(splits(0), splits(1).toLong), splits(2).toInt)
})

//按照主键分区,产生Partitioner对象 ❷
val partitioner = new Partitioner {
  override def numPartitions: Int = numberOfPartitions

  override def getPartition(key: Any): Int = {
    Math.abs(key.asInstanceOf[DataKey].uniqueId.hashCode() % numPartitions)
  }
}

//分区和排序 ❸
val partedSortedRDD =
  new ShuffledRDD[DataKey, Int, Int](
    originalDataRDD,
    partitioner).setKeyOrdering(implicitly[Ordering[DataKey]])

//使用MapPartition执行开窗操作 ❹
val pivotPointRDD = partedSortedRDD.mapPartitions(it => {

  val results = new mutable.MutableList[PivotPoint]

  //保存上下文 ❺
  var lastUniqueId = "foobar"
  var lastRecord: (DataKey, Int) = null
  var lastLastRecord: (DataKey, Int) = null

  var position = 0

  it.foreach( r => {
    position = position + 1

    if (!lastUniqueId.equals(r._1.uniqueId)) {

      lastRecord = null
      lastLastRecord = null
    }

    //查找定位数据高峰与谷底
    if (lastRecord != null && lastLastRecord != null) { ❻
      if (lastRecord._2 < r._2 && lastRecord._2 < lastLastRecord._2) {
        results.+=(new PivotPoint(r._1.uniqueId,
          position,
          lastRecord._1.eventTime,
          lastRecord._2,
          false))
      } else if (lastRecord._2 > r._2 && lastRecord._2 > lastLastRecord._2) {
```

```
            results.+=(new PivotPoint(r._1.uniqueId,
              position,
              lastRecord._1.eventTime,
              lastRecord._2,
              true))
          }
        }
        lastUniqueId = r._1.uniqueId
        lastLastRecord = lastRecord
        lastRecord = r

      })

      results.iterator
    })

    //格式化输出
    pivotPointRDD.map(r => {  ❼
      val pivotType = if (r.isPeak) "peak" else "valley"
      r.uniqueId + "," +
        r.position + "," +
        r.eventTime + "," +
        r.eventValue + "," +
        pivotType
    } ).saveAsTextFile(outputPath)

  }


  class DataKey(val uniqueId:String, val eventTime:Long)
    extends Serializable with Comparable[DataKey] {
    override def compareTo(other:DataKey): Int = {
      val compare1 = uniqueId.compareTo(other.uniqueId)
      if (compare1 == 0) {
        eventTime.compareTo(other.eventTime)
      } else {
        compare1
      }
    }
  }

  class PivotPoint(val uniqueId: String,
                   val position:Int,
                   val eventTime:Long,
                   val eventValue:Int,
                   val isPeak:Boolean) extends Serializable {}

}
```

❶ 这一段代码没什么太多要说的，就是读取输入数据，将其解析成容易处理的对象。

❷ 代码从这里真正开始变得有趣。我们定义了一个 `Partition`，这与在 MapReduce 编程中创建一个自定义 `Partitioner` 类似。`Partition` 的作用是在 Shuffle 处理后确定哪一条记录分配到哪一个 Worker。在这里，我们需要自定义 Partitioner，因为 `key` 是由 `primary_`

key 和 position 两部分组成的。我们希望排序能够同时考虑两者，但只按照 primary_key 分区。因此，输出如图 4-3 所示。

❸ 这部分执行 Shuffle 操作，分区并对数据排序。注意，Spark 1.3 提供了一个名为 repartitionAndSortWithinPartitions() 的变换方法，能够完成我们所需的功能。但是我们使用的是 Spark 1.2 的版本，需要手动实现这一 Shuffle 过程。

❹ 在 mapPartition() 方法中，primary_key 会根据位置的先后顺序进行遍历。这里也是发生数据开窗的地方。

❺ 为了找到数据的高峰和低谷，并确定是否需要更换 primary_key，我们将上下文信息保存于此。注意，为了查找数据的高峰和低谷，我们需要知道当前记录前面与后面的记录。因此，这里有如下变量：currentRow、lastRow、lastLastRow，我们可以通过比较 lastRow 与其他两个变量的值，确定 lastRow 是高峰还是低谷。

❻ 进行比较，确定是否发现了数据的高峰或低谷。

❼ 这是最后一部分代码，将记录格式化并写入 HDFS 中。

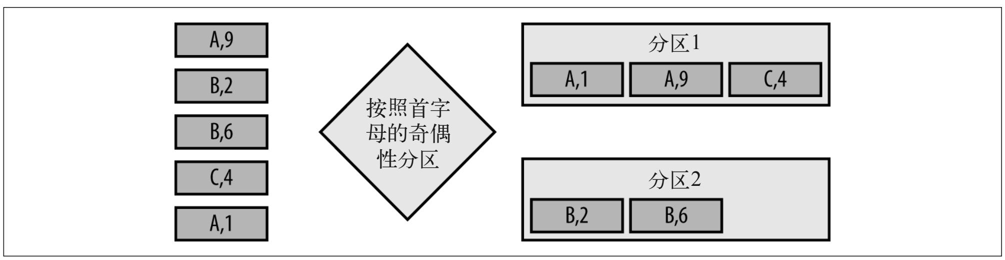


图 4-3：数据高峰及低谷中的分区示例。在这里，我们将多个数据序列分成了两个组，这样就可以将分析的工作分布到两个 Worker 上处理，每个序列对应的事件分布在一个组之内

4.2.3 代码示例：使用SQL分析数据的高峰和低谷

类似于前面的例子，我们首先基于测试数据创建如下一张数据表。

```
CREATE EXTERNAL TABLE PEAK_AND_VALLEY_TABLE (
  PRIMARY_KEY STRING,
  POSITION BIGINT,
  EVENT_VALUE BIGINT
)
ROW FROMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 'PeakAndValleyData';
```

拥有了所需的数据表之后，接下来要对记录排序，并使用 lead() 和 lag() 函数获取当前记录的上下文记录。

```
SELECT                         ❶
    PRIMARY_KEY,
    POSITION,
    EVENT_VALUE,
```

```
    CASE
     WHEN LEAD_EVENT_VALUE is null or LAG_EVENT_VALUE is null THEN 'EDGE'
     WHEN
       EVENT_VALUE < LEAD_EVENT_VALUE AND EVENT_VALUE < LAG_EVENT_VALUE
     THEN
       'VALLEY'
     WHEN
       EVENT_VALUE > LEAD_EVENT_VALUE AND EVENT_VALUE > LAG_EVENT_VALUE
     THEN
       'PEAK'
     ELSE'SLOPE'
    END AS POINT_TYPE
  FROM
   (
    SELECT                      ❷
     PRIMARY_KEY,
     POSITION,
     EVENT_VALUE,
     LEAD(EVENT_VALUE, 1, null)
          OVER(PARTITION BY PRIMARY_KEY ORDER BY POSITION) AS LEAD_EVENT_VALUE,
     LAG(EVENT_VALUE, 1, null)
          OVER(PARTITION BY PRIMARY_KEY ORDER BY POSITION) AS LAG_EVENT_VALUE
    FROM PEAK_AND_VALLEY_TABLE
   ) A
```

虽然这句 SQL 并非过度复杂，但仍然需要拆成两部对其解释。

❶ 在执行了第二步的子查询之后，我们得到了组织好的数据。一条记录中包含了需要的全部信息。我们可以凭这条记录确认以下几种类型：边缘（时间窗口的最左或最右一个点）、高峰（前后的值均比当前值要小）、低谷（前后的值均比当前值要大）、斜坡（前后的值要么都比当前值大，要么都比当前值要小）。

❷ 在这条子查询中，我们执行数据开窗的逻辑。该查询会将当前值之前和之后的值放入同一行。图 4-4 所示就是这条子查询的输入和输出。

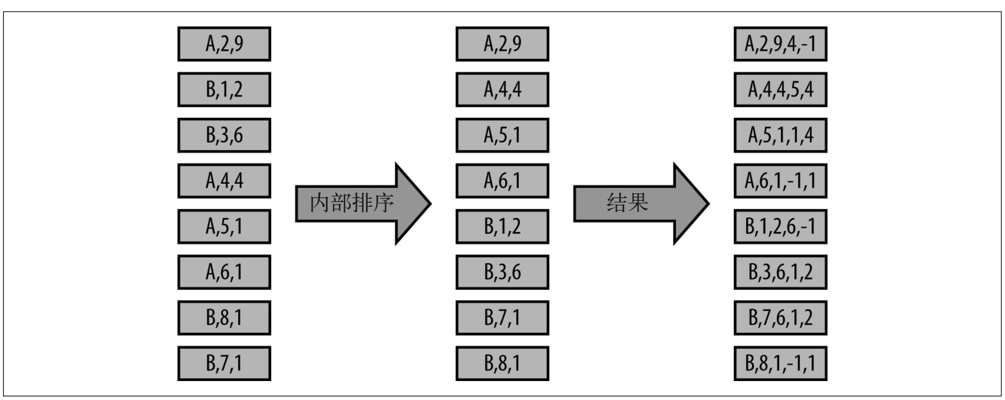


图 4-4：前述子查询的输入及输出：第一步将事件分成多个组，每个子序列一个组，并按照事件发生的顺序进行组内排序；第二步对每一个事件进行扩容，加上之前和之后发生的事件的值，以便后续检测高峰和低谷

窗口函数与 SQL

本例中的 SQL 代码比 Spark 的实现更为简洁，大家可能会觉得使用 SQL 比较简单。但我们仍然需要考虑到 SQL 的局限性。使用 SQL 执行多次复杂的窗口操作，意味着提升读写磁盘的复杂度，即增加了磁盘 I/O，同时也降低了性能。使用 Spark 则只需对数据进行一次排序，可以借助 Java 或 Scala 提供的功能执行 *N* 次窗口操作，将信息保存在内存中并执行操作。

4.3　模式三：基于时间序列的更新

本章最后一个例子会把前面两个示例结合起来。在这一个例子中，我们基于主键更新记录，并保留全部的历史信息。

这允许一条记录了解自己的生效时间及失效时间，提供一个实体的信息以及开始和结束的时间，如图 4-5 所示。

价格表

PKey	开始时间	结束时间	值
Apple	12/01/2014	null	42
Apple	11/25/2014	12/01/2014	41
Apple	11/20/2014	11/25/2014	40

图 4-5：一段时间内 Apple 公司股票的价格

开始和结束时间能够标识一条记录生效的时间区间。如果结束时间为 `null`，这就意味着该记录的值就是实体的当前值。在上面的例子中，我们可以看到 Apple 公司股票的当前价格为 42 美元。

随之而来的问题是何时更新表。如图 4-6 所示，需要添加一条价格为 43 美元的新记录时，我们需要更新前一条记录的结束时间，并将其改为新记录的开始时间。图中颜色较深的一个单元代表的是需要更新的记录。

价格表

PKey	开始时间	结束时间	值
Apple	12/05/2014	null	43
Apple	12/01/2014	12/05/2014	42
Apple	11/25/2014	12/01/2014	41
Apple	11/20/2014	11/25/2014	40

图 4-6：向 Apple 公司股票价格表中添加一条新记录

表面上看，这似乎是一个很简单的问题。实际上海量数据集的更新在实现上可能会非常复杂。接下来的讨论会涉及一些挑战和解决方案。

4.3.1　利用HBase的版本特性

存储此类信息的一种常见方式就是将记录存成 HBase 的多个版本。HBase 支持对记录的更改以多个版本的形式存储。版本在列这一层定义，并按照更新的时间戳有序排列。因此，你可以查看任意时间点的记录值。

这样做的优点是更新速度快，能够快速获取最新的值。然而，获取历史版本的性能会有一些损失，在执行大型扫描操作或读数据块缓存时，性能缺陷较为明显。

大型的扫描操作较慢，这是因为在到达下一个记录前，每条记录的所有版本都要读取。如果版本数很多，那么扫描操作会非常慢。

读取数据块缓存时，命中率不高，因为在读取一条记录时，HBase 会将整个 64 KB 的 HFile 数据块缓存于内存中。如果历史信息很多，那么缓存中的数据很有可能是记录的其他版本，而不是你实际想要的记录。

最后，这个数据模型并没有将开始和结束时间保存在同一条记录中。想要了解一个版本的开始和结束时间，你需要参照多个版本记录。

4.3.2　以记录主键与开始时间作HBase的行键

为了在 HBase 的同一条记录中保存开始和结束时间，我们将创建一个由 `RecordKey` 和 `StartTime` 构成的组合键。与之前提到的多版本解决方案不同，这个新方案的记录会有一个列用来存储结束时间。

注意，`RowKey` 中的开始时间需要将时间戳反转，以保证按照从新到旧的顺序排列。

因此，增加一条新版本的记录时，这里需要进行如下修改。首先，以 `RecordKey` 作为起始 `RowKey` 进行单行扫描，得到指定 `RecordKey` 对应的当前版本的数据。拥有这一信息之后，接下来要执行两次 `put` 操作：先更新当前记录的结束时间，然后再添加一个新的当前记录。

获得指定时间版本的记录时，你只需要以 `RecordKey` 和指定时间的反转值为行键进行单行扫描即可。

这一方案在大型数据扫描和数据块缓存方面存在一定的问题，但是可以快速地获取某一版本，并在同一行中保存结束时间。值得注意的是，这需要在插入一条记录时外加 `get` 和 `put` 操作。

4.3.3　重写HDFS数据更新整个表

如果放弃使用 HBase，只使用 HDFS 作为实现方案，那么我们需要将所有的数据存储到 HDFS 上。按照某个特定的周期（如一天执行一次）获取新数据时，刷新数据表即可。

这一方案看起来代价很大。不过，借助 Hadoop 我们可以在较短的时间内重写 TB 级的数据。随着数据集的增大，你还可以借助一些技术，优化这一执行过程。举例来说，你可以

为大多数的当前记录建立单独的分区。后面我们会继续讨论这一方案。

4.3.4 利用HDFS上的分区存储当前记录和历史记录

在 HDFS 上，这是一个较为明智的做法：将大多数存储当前值的记录放到一个分区里，将其他历史记录放置于另外一个分区中。这样可以只重写最新的版本，而不涉及历史版本。接下来将新记录追加到旧记录所在的分区即可。

这样做的最大好处在于执行时间基本固定，不会随历史版本数据的增加而拉长。

接下来，我们就这种方式讲解一个例子，同样是分别使用 Spark 和 SQL 两种工具来实现。出于演示的目的，我们使用一个简单的、模拟的数据集，每条记录由唯一的 ID、事件发生时间和随机整数组成。不过，这个例子中使用的方法很容易扩展到真实场景中基于事件的数据（如之前描述的股票报价）。

4.3.5 生成时间序列的示例数据

我们在这里根据示例的需要，创建基于时间序列的数据集，同样使用 Scala 和 HDFS 文件系统的 API。

```
def main(args: Array[String]): Unit = {
  if (args.length == 0) {
    println("{outputPath} {numberOfRecords} {numberOfUniqueRecords} {startTime}")
    return
  }

  val outputPath = new Path(args(0))
  val numberOfRecords = args(1).toInt
  val numberOfUniqueRecords = args(2).toInt
  val startTime = args(3).toInt

  val fileSystem = FileSystem.get(new Configuration())
  val writer =
      new BufferedWriter(new OutputStreamWriter(fileSystem.create(outputPath)))

  val r = new Random


  for (i <- 0 until numberOfRecords) {
    val uniqueId = r.nextInt(numberOfUniqueRecords)
    val madeUpValue = r.nextInt(1000)
    val eventTime = i + startTime

    writer.write(uniqueId + "," + eventTime + "," + madeUpValue)
    writer.newLine()
  }
  writer.close()
}
```

通读本节的数据生成代码，你会发现它的设计思路与之前例子中的数据生成代码极为类似。

4.3.6 代码示例：使用Spark更新时间序列数据

现在，让我们关注 Spark 代码的具体实现，了解如何按照时间序列数据的起止时间进行单分区的更新。你可以在 SparkTimeSeriesExecution 这一 Scala 对象的 GitHub 中找到代码片段。这是本章最长的示例代码，我们稍后会进行分析。

```
object SparkTimeSeriesExecution {
  def main(args: Array[String]): Unit = {
    if (args.length == 0) {
      println("{newDataInputPath} " +
        "{outputPath} " +
        "{numberOfPartitions}")
      println("or")
      println("{newDataInputPath} " +
        "{existingTimeSeriesDataInputPath} " +
        "{outputPath} " +
        "{numberOfPartitions}")
      return
    }

    val newDataInputPath = args(0)
    val existingTimeSeriesDataInputPath = if (args.length == 4) args(1) else null
    val outputPath = args(args.length - 2)
    val numberOfPartitions = args(args.length - 1).toInt

    val sparkConf = new SparkConf().setAppName("SparkTimeSeriesExecution")
    sparkConf.set("spark.cleaner.ttl", "120000");

    val sc = new SparkContext(sparkConf)

    //从HDFS中加载数据
    var unendedRecordsRDD = sc.hadoopFile(newDataInputPath,   ❶
      classOf[TextInputFormat],
      classOf[LongWritable],
      classOf[Text],
      1).map(r => {
      val splits = r._2.toString.split(",")

      (new TimeDataKey(splits(0), splits(1).toLong),
        new TimeDataValue(-1, splits(2)))
    })


    var endedRecordsRDD:RDD[(TimeDataKey, TimeDataValue)] = null

    //若存在,则读取已有的记录
    if (existingTimeSeriesDataInputPath != null) {      ❷
      val existingDataRDD = sc.hadoopFile(existingTimeSeriesDataInputPath,
        classOf[TextInputFormat],
        classOf[LongWritable],
        classOf[Text],
        1).map(r => {
        val splits = r._2.toString.split(",")
        (new TimeDataKey(splits(0), splits(1).toLong),
```

```
      new TimeDataValue(splits(2).toLong, splits(3)))
  })

  unendedRecordsRDD = unendedRecordsRDD
    .union(existingDataRDD.filter(r => r._2.endTime == -1))

  endedRecordsRDD = existingDataRDD.filter(r => r._2.endTime > -1)
}

//定义本例中所需的Partitioner
val partitioner = new Partitioner {   ❸
  override def numPartitions: Int = numberOfPartitions

  override def getPartition(key: Any): Int = {
    Math.abs(
      key.asInstanceOf[TimeDataKey].uniqueId.hashCode() % numPartitions)
  }
}

val partedSortedRDD =
  new ShuffledRDD[TimeDataKey, TimeDataValue, TimeDataValue](
    unendedRecordsRDD,
    partitioner).setKeyOrdering(implicitly[Ordering[TimeDataKey]])

//遍历每一个主键,确保停止时间的值已更新
var updatedEndedRecords = partedSortedRDD.mapPartitions(it => {   ❹
  val results = new mutable.MutableList[(TimeDataKey, TimeDataValue)]

  var lastUniqueId = "foobar"
  var lastRecord: (TimeDataKey, TimeDataValue) = null

  it.foreach(r => {
    if (!r._1.uniqueId.equals(lastUniqueId)) {
      if (lastRecord != null) {
        results.+=(lastRecord)
      }
      lastUniqueId = r._1.uniqueId
      lastRecord = null
    } else {
      if (lastRecord != null) {
        lastRecord._2.endTime = r._1.startTime
        results.+=(lastRecord)
      }
    }
    lastRecord = r
  })
  if (lastRecord != null) {
    results.+=(lastRecord)
  }
  results.iterator
})

//并入已存在的记录
if (endedRecordsRDD != null) {   ❺
  updatedEndedRecords = updatedEndedRecords.union(endedRecordsRDD)
```

```
    }

    //格式化结果,并保存至HDFS
    updatedEndedRecords    ❻
      .map(r => r._1.uniqueId + "," +
      r._1.startTime + "," +
      r._2.endTime + "," +
      r._2.data)
      .saveAsTextFile(outputPath)
  }

  class TimeDataKey(val uniqueId:String, val startTime:Long)
      extends Serializable with Comparable[TimeDataKey] {
    override def compareTo(other:TimeDataKey): Int = {
      val compare1 = uniqueId.compareTo(other.uniqueId)
      if (compare1 == 0) {
        startTime.compareTo(other.startTime)
      } else {
        compare1
      }
    }
  }

  class TimeDataValue(var endTime:Long, val data:String) extends Serializable {}
}
```

❶ 与之前的示例一样，这一部分从 HDFS 读取新的数据集。

❷ 这一部分与本章之前的示例不同。在本例中，输入可能是新的数据，也可能是 HDFS 中已有表中的数据。在这里，我们认为 HDFS 表中的数据集未必存在，因为显然在第一次添加记录时，HDFS 表中的数据集是空的。注意，我们会过滤掉拥有 `endTime` 值的记录，这是因为我们不希望这类数据经过后续的 Shuffle 过程，并通过网络传输以及执行排序等操作。后面我们会合并这些值。

❸ 与数据高峰及低谷的示例一样，我们需要一个自定义的 `Partition` 和 Shuffle。只要数据集需要按照指定 `key` 排序，类似的处理方式就会用到。这里的分区策略与先前一致：按照 `primaryKey` 分区，按照 `primaryKey` 和 `startTime` 的组合排序。

❹ 在这里遍历每一个 `primaryKey`，更新记录对应的停止时间。

❺ 这一步我们使用 `union()` 方法，将拥有 `endTime` 的已有记录合并到最终结果中。

❻ 最后，我们将格式化的结果输出到 HDFS 上。

4.3.7　代码示例：使用SQL更新时间序列数据

与之前类似，我们要先建设数据集的源表，供 Hive 或 Impala 使用。这个例子会有两张表，一个是已存在的时间序列表，另一个是新增表。

```
CREATE EXTERNAL TABLE EXISTING_TIME_SERIES_TABLE (
  PRIMARY_KEY STRING,
  EFFECTIVE_DT BIGINT,
  EXPIRED_DT BIGINT,
```

```
  EVENT_VALUE STRING
  )
 ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
 STORMED AS TEXTFILE
 LOCATION 'ExistingTimeSeriesData';

CREATE EXTERNAL TABLE NEW_TIME_SERIES_TABLE (
  PRIMARY_KEY STRING,
  EFFECTIVE_DT BIGINT,
  EVENT_VALUE STRING
  )
 ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
 STORMED AS TEXTFILE
 LOCATION 'NewTimeSeriesData';
```

这两张表非常类似，只是后者没有失效日期这一字段。

注意，在实际应用场景中，我们会对数据集进行分区，比如为当前记录和历史记录分区。这是因为我们不必读取那些已经拥有失效日期的记录，要执行的操作不需要这些数据。我们只需读取那些已有的活跃记录，根据 NEW_TIME_SERIES_TABLE 表中的新增记录来判断它们是否失效。

另外，当前的记录会被重写，而针对历史记录则只会进行追加操作。

现在，让我们根据这两张表生成一个包含更新后记录的结果表。

```
SELECT
    PRIMARY_KEY,
    EFFECTIVE_DT,
    CASE
      WHEN LEAD(EFFECTIVE_DT, 1, null)
        OVER
          (PARTITION BY PRIMARY_KEY ORDER BY EFFECTIVE_DT)
             IS NULL THEN NULL
      ELSE LEAD(EFFECTIVE_DT, 1, null)
        OVER
          (PARTITION BY PRIMARY_KEY ORDER BY EFFECTIVE_DT)
    END AS EXPIRED_DT,
    EVENT_VALUE
   FROM (
     SELECT
       PRIMARY_KEY,
       EFFECTIVE_DT,
       EVENT_VALUE
     FROM
       EXISTING_TIME_SERIES_TABLE
     WHERE
       EXPIRED_DT IS NULL
     UNION ALL
     SELECT
       PRIMARY_KEY,
       EFFECTIVE_DT,
       EVENT_VALUE
```

```
    FROM NEW_TIME_SERIES_TABLE
    ) sub_1
UNION ALL
SELECT
  PRIMARY_KEY,
  EFFECTIVE_DT,
  EXPIRED_DT,
  EVENT_VALUE
FROM
  EXISTING_TIME_SERIES_TABLE
WHERE
  EXPIRED_DT IS NOT NULL
```

这是一个相当长的查询语句，我们分解开来看。最外层是两个 SELECT 语句联合在一起。前面一个 SELECT 语句处理已存在的当前记录和新增的记录，判断哪些是失效的记录。后面一个 SELECT 语句则挑选出已经失效的记录。如果使用了前面提到的分区策略，那么外层的第二个 SELECT 语句就不需要了。

这里重点讲解第一个 SELECT 语句。如下面的代码片段所示，子查询中有两个表的联合。我们过滤了 EXISTING_TIME_SERIES_TABLE 表中的失效日期的记录，因为我们不需要对这些数据进行排序和窗口函数操作。

```
SELECT
  PRIMARY_KEY,
  EFFECTIVE_DT,
  EVENT_VALUE
FROM
  EXISTING_TIME_SERIES_TABLE
WHERE
  EXPIRED_DT IS NULL
UNION ALL
  SELECT
    PRIMARY_KEY,
    EFFECTIVE_DT,
    EVENT_VALUE
  FROM NEW_TIME_SERIES_TABLE
```

内部 SELECT 生成的结果会按照主键分区，并按照生效日期和时间排序。在这里，我们会考虑这样一个问题："确定一个主键后，这条记录是否存在更大的时间戳，并包含生效日期？"如果答案是肯定的，那么这条记录就是失效记录。以下是查询的代码片段。

```
PRIMARY_KEY,
EFFECTIVE_DT,
CASE
  WHEN LEAD(EFFECTIVE_DT, 1, null)
    OVER
      (PARTITION BY PRIMARY_KEY ORDER BY EFFECTIVE_DT)
          IS NULL THEN NULL
  ELSE LEAD(EFFECTIVE_DT, 1, null)
    OVER
      (PARTITION BY PRIMARY_KEY ORDER BY EFFECTIVE_DT)
END AS EXPIRED_DT,
EVENT_VALUE
```

如你所见，这的确是一个较为复杂的查询语句，但是与 Spark 的代码比起来，还是更为简洁。在这种情况下，Spark 或者 SQL 都是不错的选择，选择哪一个取决于你对工具的喜好与熟悉程度。

4.4 小结

总结一下，本章得出了以下几条结论。

- 在 Hadoop 上使用 Spark 和 SQL 可以进行一些比较复杂的处理。
- 迁移到 Hadoop 上时，你不必弃用 SQL。实际上，SQL 在 Hadoop 平台上仍然是强大的数据处理和数据分析抽象。这一点与传统数据管理系统的情况一致。
- 我们可以采用不同的工具解决问题，不同的工具也会给我们不同的解决方案。

随着新型工具（如 Impala 和 Hive on Spark）的引入，SQL-on-Hadoop 工具会越来越强大。不过在解决各类问题的时候，SQL 并不会取代 Spark 及其他处理框架。全面发展的 Hadoop 开发者需要了解已有工具，并针对特定问题评估和确定最合适的工具。

第 5 章

Hadoop图处理

在第 3 章中，我们讨论了用于 Hadoop 数据处理的工具，但是没有讨论另一种类型的 Hadoop 数据处理：图数据处理。现在，图数据处理成为了很多应用的核心，而且为特定类型的应用实现提供了重要的机会，所以我们将在本章单独讨论 Hadoop 图数据处理。一些常用的图数据处理案例为搜索引擎中的网页排名、在社交网络上查找新朋友、在投资基金中查找确定具有潜力的股票，以及路径规划，等等。

5.1　什么是图

说不定你是第一次接触图数据，所以在讨论用于图数据处理的工具之前，我们先来看一下图数据的定义。接下来我们会了解什么是图数据处理，以及图数据处理与图数据查询的区别。

无需赘言，“图”可以表示很多含义。它可以指 Excel 工具中的折线图、饼图或者柱状图。本章要讨论的图并不是花花绿绿的图画。我们要讨论的图只包含两种要素，即顶点（vertex）和边（edge）。

你可能在上学的时候就已经学过，三角形的尖被称作顶点，而连接两个点之间的线被称作边，如图 5-1 所示。对图有了些许认识之后，让我们开始进一步学习。

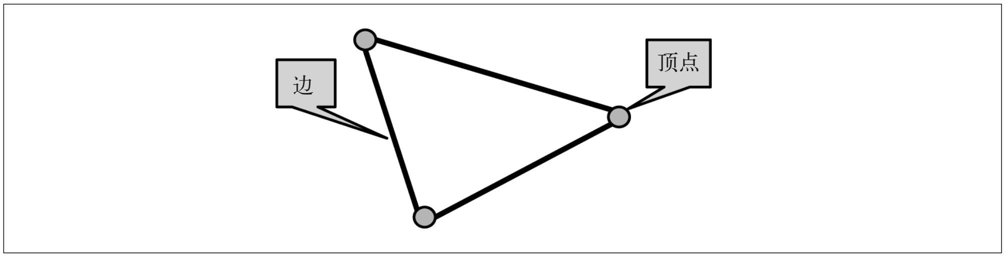


图 5-1：边与顶点

现在我们稍微改变一下上图的内容，给每个点都添加一些信息。假设每个点代表一个人，而我们想知道这个人的一些信息，如他的名字以及类别（这种情况下为 `Person`）。现在，我们也给每个边添加一些信息，描述相连两点之间的关系。这些信息可以是看过的电影，或者兄弟、父亲、母亲以及妻子之类的关系（如图 5-2）。

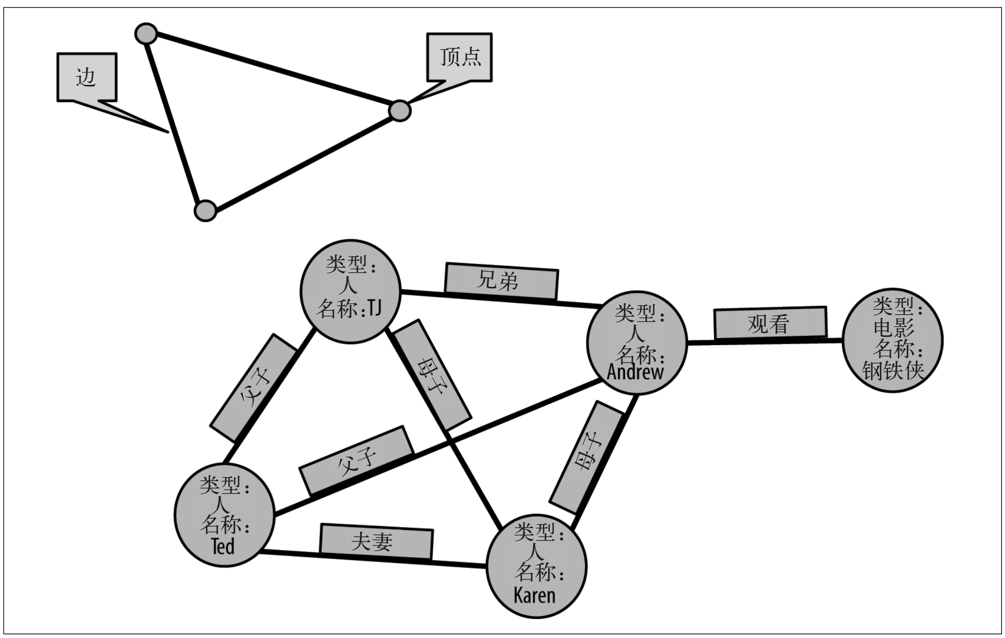


图 5-2 ：将信息附加到点和边上

但是这些信息仍然不够。我们知道 Karen 是 TJ 的母亲，但是 TJ 不可能也是 Karen 的母亲，而且我们知道 Andrew 看过电影《钢铁侠》，但是《钢铁侠》不可能看 Andrew。所以，我们可以给每个边定一个方向，这样就解决了上述问题。效果如图 5-3 所示。

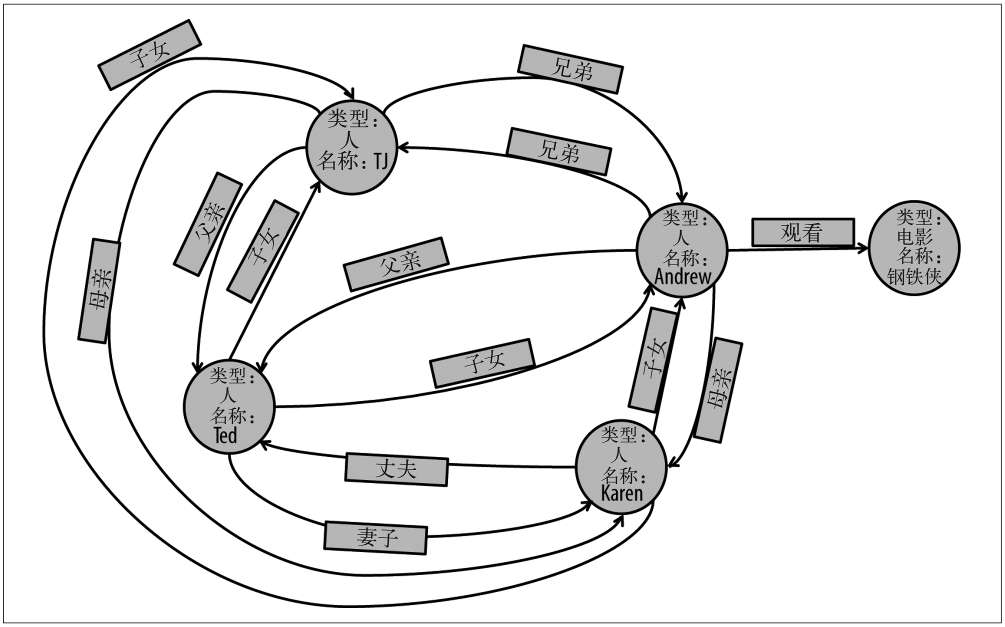


图 5-3 ：描述点之间有向关系的边

这样就好了。即使我们把这张图拿给一个毫无技术背景的人看，他也能够理解我们想要表达的内容。

5.2 什么是图处理

在图数据处理中，我们所说的是全局层面的处理，可能会遍历图中的每个点。这与*图查询*（graph querying）所包含的意思有所不同。图数据的查询在用户提出问题（比如，“谁与 Karen 有关系？”）时才会执行。该类查询按照以下步骤执行。

(1) 查找代表 Karen 的点，以及与该点相连的所有的边。
(2) 遍历每一条边得到相应的点。
(3) 返回给用户结果列表。

在这里，*图数据*处理所包含的意思稍有不同。我们指的不是“找出五度人脉中的 Top 5”。与图的查询相比，这个问题涉及寻找大量的用户及其之间的关系，处理的数据规模更大，需要更多的处理资源。与图数据处理形成对比的是，图的查询可以通过一个客户端访问类似于 HBase 的数据存储，只需若干个跳转查询就可以完成对单个用户的查询。

这些概念在实际案例中的应用如下所示。

- 你可以把网络看作一个非常大的图，网页是点，超链接是边。你可以就此进行各种分析。Google 用于计算搜索结果排名的著名算法 PageRank 就是一个例子。
- 如前面内容所述，社交网络是一种天然的图数据处理应用。比如，你可以就此测定一个网站中各用户之间的人脉度数。

本章要关注的重点就是这些类型的应用。那么，如何利用 Hadoop 数据高效地使用可维护的、性能较高的方法来解决这类问题呢？

5.3 分布式系统中的图处理

为了在类似 Hadoop 之类的系统中执行这类数据处理，我们先从 MapReduce 开始。问题是 MapReduce 只能提供一层合并，这表明我们不得不像剥洋葱一样来处理图数据。对于不剥洋葱的人来说，这很新鲜，因为剥洋葱和削苹果完全不同。洋葱只有剥掉很多层之后才能看到核心。另外，洋葱会让人流眼泪。洋葱的刺激性让剥洋葱的过程更加不愉快，使用 MapReduce 进行图数据处理也是这样的。MapReduce 有时可能会让你想哭。图 5-4 中的图为一个类似于“剥洋葱”的使用案例。中点为最开始的人，而每个增加的圈都是另一个 MapReduce 任务，每层中相连的人就是这样计算的。

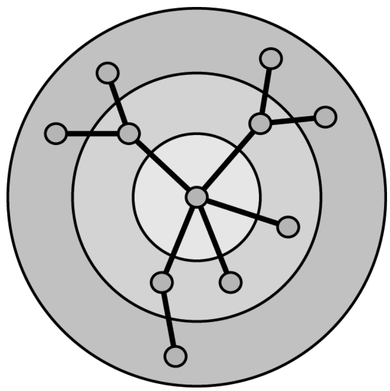


图 5-4 ：在 MapReduce 中，图处理如同剥洋葱

意识到每一条路径都需要重新将整个图读写到磁盘，你会更加痛苦。

幸好，Google 的智者再一次决定打破这个规则，改变 MapReduce 框架中 mapper 之间不能通信的情况。这种无共享的思路对于 Hadoop 这样的分布式系统来说非常重要，因为它们依赖于同步点和故障恢复策略。聪明人是怎么解决这个问题的呢？总之他们找到了另外的方法应对同步点和恢复策略，而不会受到 mapper 的限制。

5.3.1 块同步并行模型

那么，我们怎样才能保持同步处理而又打破“mapper 之间不能通信”的规则呢？来自哈佛大学的英国计算机科学家 Leslie Valiant 提供了一个解决方案，他在 20 世纪 90 年代创建了块同步并行模型（Bulk Synchronous Parallel，BSP）。BSP 模型正是 Google 图处理解决方案 Pregel 的核心。

BSP 的理念非常复杂，同时又很简单，说穿了就是在一个 `superstep` 之内执行分布式的数据处理任务。这种分布式数据处理进程能够互相发送消息，但是直到下一个 `superstep` 开

始之前都不能处理消息。这些 superstep 将作为我们需要的同步点发挥作用。所有的分布式处理任务都完成，并且当前 superstep 的消息也已经发送完毕后，才会到达下一个 superstep。接下来通常会有一个单线程程序，决定整个数据处理是否需要继续并进入新的 superstep。与工作任务线程相比，它需要执行的任务非常少，所以可以接受该处理在单线程中运行，而不会成为瓶颈。

5.3.2 BSP举例

不可否认，确定分布式数据处理的简短定义花费了数年的研究。我们将通过一个 BSP 案例来佐证和分析相关概念。科学家可以使用图处理的方法，对疾病在一个社区内的传播建立模型。本案例中，我们通过僵尸来阐明这一概念——僵尸原本是人类，被别的僵尸咬过之后变为僵尸。

让我们来制作一张新的图，并把这个图命名为僵尸咬人。如图 5-5，开始时图中含有一个僵尸和一群人。数据处理开始时规则是这样的：僵尸可以咬每一个跟其共享同一个边的人。当一个点被咬之后，这个点上的人转变为僵尸，而且开始咬与其相连接的其他人。一旦咬人，僵尸周围的人就变成了僵尸，所以这个僵尸不会再继续咬周围的点。有无数的僵尸电影能够告诉我们，僵尸不咬其他僵尸。

图 5-5 为 BSP 执行模型的不同 superstep 中的图表。

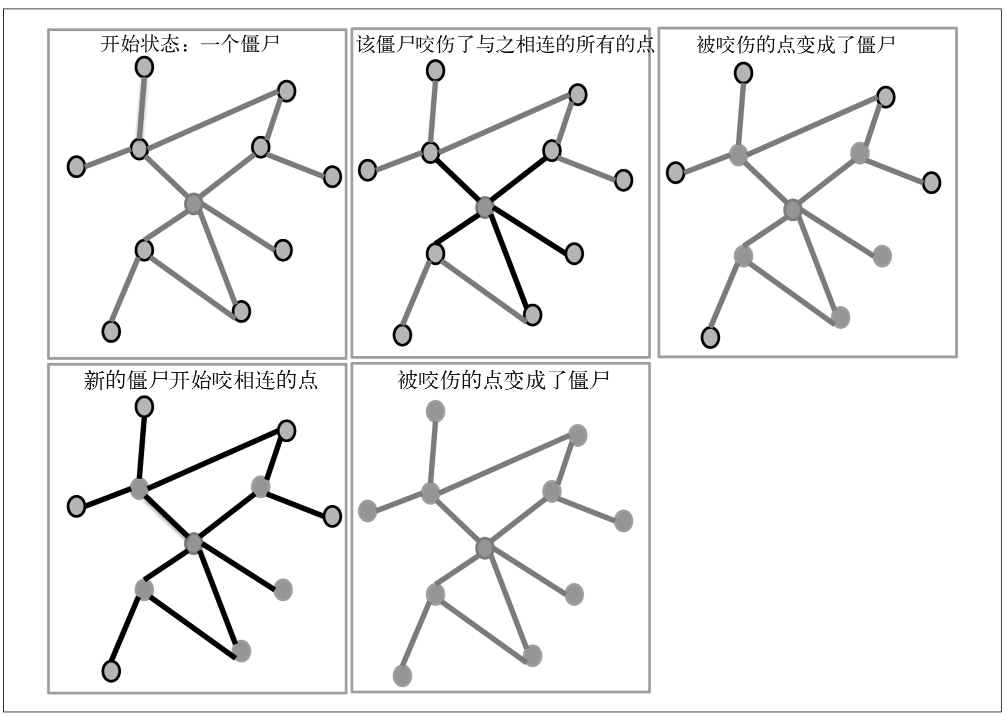


图 5-5：僵尸咬人图的 superstep 示意图

本章将介绍两个图数据处理工具（Giraph 与 GraphX），并演示该案例的实现。但是在这之前你需要明白，BSP 并不是唯一的解决方法。我们在深入讨论 Spark 时就了解到，使用 Spark 能够在很大程度上降低 MapReduce 那种剥洋葱式方法带来的缺陷。MapReduce 使得每个洋葱分层之间的 I/O 读写操作频繁，而 Spark 大大缓解了这些问题，至少在数据能够存储在内存时效果显著。但是我们也已经说过，BSP 模型非常独特，只会在分布式数据处理进程之间发送消息。使用剥洋葱的方法则会重新发送所有的消息。

在后面两节，我们将深入讨论目前两个最常用的 Hadoop 图数据处理框架。首先要介绍的是 LinkedIn 创建，由 Facebook 用于图数据搜索的 Giraph。Giraph 是一种更为成熟稳定的系统，宣称能处理多达一万亿的边。

第二个工具是较为年轻的 GraphX，它是 Apache Spark 项目的一部分。Spark GraphX 的创建基础主要为 GraphLab（http://dato.com/products/create/open_source.html），这是一个早期的开源图数据处理项目，在 Spark 通用 DAG 执行引擎的基础上扩展而成。尽管 GraphX 仍然是一种很年轻的工具，而且不如 Giraph 灵活稳定，但它的使用方法简单，而且能够与 Spark 的所有其他组件兼容，所以 GraphX 的前景很广阔。

5.4 Giraph

Giraph 是 Google Pregel 的一种开源实现。从开始创建以来，Giraph 就被用于图数据处理。我们已经注意到这与 Spark 的 GraphX 有所不同，后者包含一种基于 Spark DAG 引擎的 Pregel API 实现。

为了简单了解一 Giraph，我们会去掉很多细节内容，并将重点放在 Giraph 项目的三个主要阶段（如图 5-6 所示）。

(1) 数据的输入和分片。
(2) 使用 BSP 进行图的批处理。
(3) 将图回写磁盘。

我们在这里没有讨论 Giraph 的其他细节。我们的目的是为你提供足够多的内容，让你能够决定在架构中添加哪种工具。

下面我们来详细了解一下这些阶段，还有自定义这些阶段所需要的代码，以及僵尸咬人的问题。

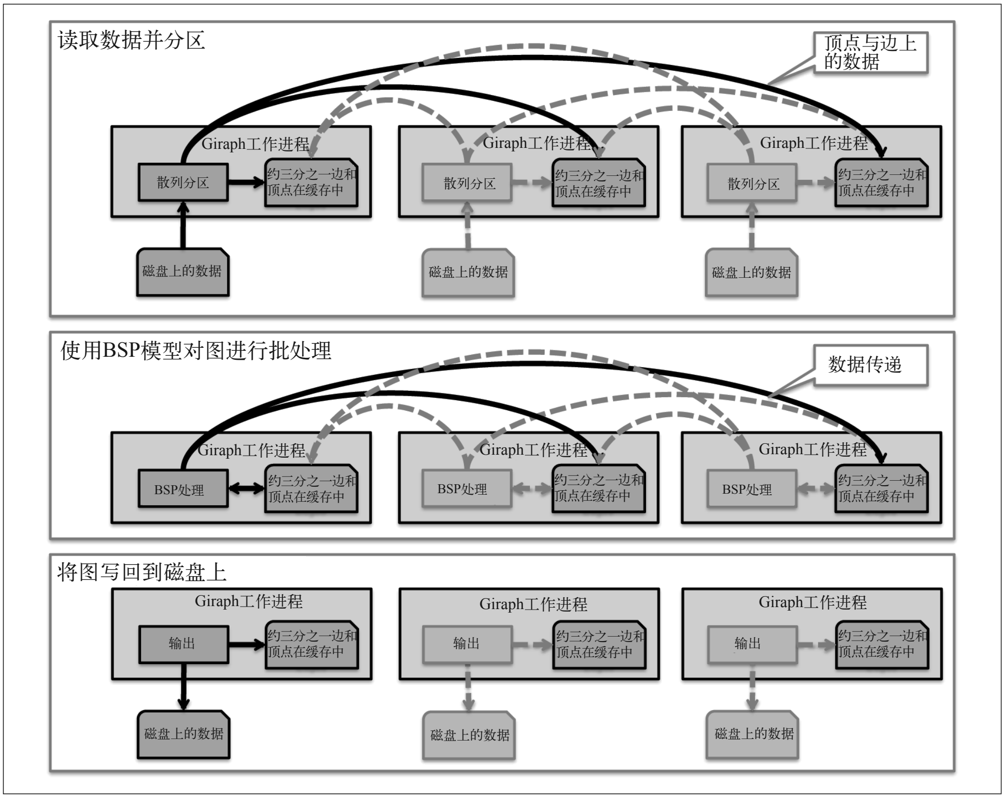


图 5-6：Giraph 程序的三个主要阶段

5.4.1　数据的输入和分片

MapReduce 与 Spark 都支持 `InputFormat`，Giraph 的输入格式 `VertexInputFormat` 则与之类似。两种情况的输入格式都能够对数据分片，并作为 Reader 获得一条记录或一个点。在这个实现中，我们将保留默认的分片逻辑，只重写 Reader。因此，`ZombieTextVertex-InputFormat` 很简单，如下所示。

```
public class ZombieTextVertexInputFormat extends
    TextVertexInputFormat<LongWritable, Text, LongWritable> {

    @Override
    public TextVertexReader createVertexReader(InputSplit split,
                                               TaskAttemptContext context)
        throws IOException {
      return new ZombieTextReader();
    }
}
```

下面我们需要一个 `VertexReader`。`VertexReader` 与普通的 MapReduce `RecordReader` 相比，主要差别在于后者会返还一个键与 `Writable` 类型的值，而前者返还一个 `Vertex` 对象。

那么，Vertex 对象又是什么呢？它主要由三部分组成。

- 顶点 ID

  这种 ID 能区分图中每一个点。

- 顶点值

  这是一个对象，包含点的信息。本例中，它将存储人或僵尸的状态，以及他或她变成僵尸的阶段。我们使用字符串“Human”表示人还未变成僵尸，使用“Zombie.2”表示在第二个 superstep 被咬。

- 边

  边由两部分组成：源的顶点 ID 与一个包含若干信息的对象。后者包含的信息代表边的方向与 / 或边的类型（比如：边代表亲戚关系、距离还是体重？）

我们已经了解了点，下面来看一下在源文件中如何描述点。

```
{vertexId}|{Type}|{comma-separated vertexId of "bitable" people}
2|Human|4,6
```

这是一个 ID 为 2 的点，目前代表人类，而且与点 4 和点 6 之间均相连着一条有方向的边。现在来看一下这条边以及将其转成 Vertex 对象的代码。

```
public class ZombieTextReader extends TextVertexReader {

  @Override
  public boolean nextVertex() throws IOException, InterruptedException {
    return getRecordReader().nextKeyValue();
  }

  @Override
  public Vertex<LongWritable, Text, LongWritable> getCurrentVertex()
      throws IOException, InterruptedException {
    Text line = getRecordReader().getCurrentValue();
    String[] majorParts = line.toString().split("\\|");
    LongWritable id = new LongWritable(Long.parseLong(majorParts[0]));
    Text value = new Text(majorParts[1]);

    ArrayList<Edge<LongWritable, LongWritable>> edgeIdList =
        new ArrayList<Edge<LongWritable, LongWritable>>();

    if (majorParts.length > 2) {
      String[] edgeIds = majorParts[2].split(",");
      for (String edgeId: edgeIds) {
        DefaultEdge<LongWritable, LongWritable> edge =
    new DefaultEdge<LongWritable, LongWritable>();
        LongWritable longEdgeId = new LongWritable(Long.parseLong(edgeId));
        edge.setTargetVertexId(longEdgeId);
        edge.setValue(longEdgeId); // dummy value
        edgeIdList.add(edge);
      }
    }

    Vertex<LongWritable, Text, LongWritable> vertex = getConf().createVertex();
```

```
      vertex.initialize(id, value, edgeIdList);
      return vertex;
    }
  }
```

这段代码包含的内容很多，下面进行分解。

- VertexReader 继承了 TextVertexReader，因此能够一行行地读取文本文件。注意，想读取任何其他类型的 Hadoop 文件，你需要改变父类的 Reader 类名。
- nextVertex() 是一种很有趣的方法。查看父类的方法，你会发现这其实使用了通用的 RecordReader 来尝试读取下一行文件，并返回剩下的文件。
- 使用方法 getCurrentVertex() 解析文件并创建、填充一个 Vertex 对象。

所以，在使用这种方法时，可以将得到的 Vertex 对象分区存储到集群中不同的分布式 Worker 中。默认的分区逻辑为基本的散列分区，你也可以对其修改。这已经超出了本例的讲解范围，我们只是在此提醒你注意控制分区。如果了解图的模式，你应该能分辨出这种情况：图分布到了较少的分布式任务中，网络使用可能不充分，性能也会有所降低。

一旦数据加载到内存或者存到磁盘中（并开启 Giraph 中新增的 spill-to-disk 功能），我们就能够开始使用 BSP 处理数据了，如 5.4.2 节所述。

在开始下一节之前，你需要注意一点：这只是一个 VertexInputFormat 的例子。Giraph 中存在更多高级的选择，如通过不同的 Reader 在点和边上读取数据，以及更为高级的分区策略，但是这些都不属于本书的讨论范围。

5.4.2 使用BSP批处理图

在 Giraph 中，对于新手来说 BSP 执行模式是最难懂的。为了让这个概念更容易理解，我们将重点放在三个计算阶段：顶点、主线程与工作线程。后面将呈现这三个阶段的代码，我们来看一下图 5-7。

从图中可以看到，每一个 BSP 都从一个主线程计算阶段开始，然后是每个分布式 JVM 的工作线程计算阶段，最后是相应的 JVM 本地内存或本地磁盘中每个点的顶点计算阶段。

这些顶点计算可能会处理消息，然后这些消息被发送到接收点，但是直到下一个 BSP 通过，接收点才能接收到这些消息。

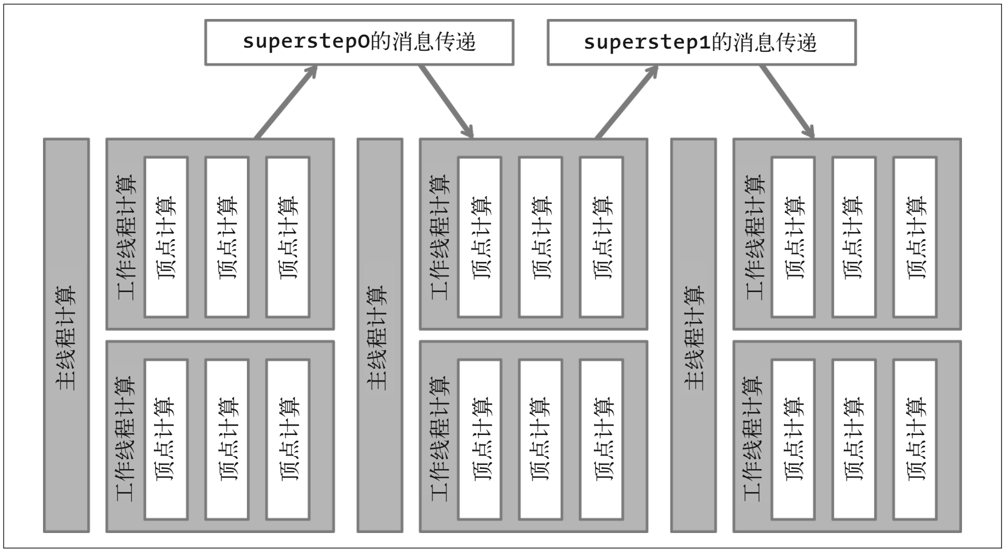


图 5-7：BSP 执行模式的三个计算阶段：顶点计算阶段、主线程计算阶段、工作线程计算阶段

我们从最简单的计算阶段（即主线程计算）开始。

```
public class ZombieMasterCompute extends DefaultMasterCompute {

  @Override
  public void compute() {
    LongWritable zombies = getAggregatedValue("zombie.count");

    System.out.println("Superstep "+String.valueOf(getSuperstep())+
                       " - zombies:" + zombies);
    System.out.println("Superstep "+String.valueOf(getSuperstep())+
                       " - getTotalNumEdges():" + getTotalNumEdges());
    System.out.println("Superstep "+String.valueOf(getSuperstep())+
                       " - getTotalNumVertices():" +
                       getTotalNumVertices());
  }

  @Override
  public void initialize()
      throws InstantiationException, IllegalAccessException {
    registerAggregator("zombie.count", LongSumAggregator.class);
  }
}
```

下面我们在 ZombieMasterCompute 类中深入研究这两个方法。我们先来看一下 initialize() 方法——在真正开始之前可以这么称呼它。重要的是我们在这里注册了一个 Aggregator 类。

Aggregator 类与 MapReduce 中的高级计数器相似，但是更像 Spark 中的 accumulator。Giraph 中存在很多 aggregator 可供选择，如后面列表所示。用户也可以创建自定义的 aggregator。

以下为一些 Giraph aggregator 的例子。

- 求和（Sum）
- 求平均（Avg）
- 求最大值（Max）
- 求最小值（Min）
- 文本方式追加（TextAppend）
- 和 / 或布尔运算（And/Or）

ZombieMasterCompute 中的第二个方法是 compute()，这种方法会在每个 BSP 开始的时候运行。在此我们只打印协助调试过程的信息。

下面讲一下后面的一些代码，即用于工作线程计算阶段的 ZombieWorkerContext。这些代码将在应用与每个 superstep 的前后执行，并且可以用于一些高级操作。比如，在一个 superstep 开始时放置合并的数据，使其能够访问顶点计算阶段。在这里，我们只使用 System.out.println()，所以我们能看到在数据处理过程中不同方法被调用的时间。

```
public class ZombieWorkerContext extends WorkerContext {

  @Override
  public void preApplication() {
    System.out.println("PreApplication # of Zombies: " +
        getAggregatedValue("zombie.count"));

  }

  @Override
  public void postApplication() {
    System.out.println("PostApplication # of Zombies: " +
        getAggregatedValue("zombie.count"));
  }

  @Override
  public void preSuperstep() {
    System.out.println("PreSuperstep # of Zombies: " +
        getAggregatedValue("zombie.count"));

  }

  @Override
  public void postSuperstep() {
    System.out.println("PostSuperstep # of Zombies: " +
        getAggregatedValue("zombie.count"));
  }
}
```

最后是最复杂的顶点计算阶段。

```
public class ZombieComputation
  extends BasicComputation<LongWritable,Text, LongWritable, LongWritable> {
  private static final Logger LOG = Logger.getLogger(ZombieComputation.class);
```

```
    Text zombieText = new Text("Zombie");
    LongWritable longIncrement = new LongWritable(1);

    @Override
    public void compute(Vertex<LongWritable, Text, LongWritable> vertex,
                        Iterable<LongWritable> messages) throws IOException {
      Context context = getContext();
      long superstep = getSuperstep();

      if (superstep == 0) {
        if (vertex.getValue().toString().equals("Zombie")) {

          zombieText.set("Zombie." + superstep);
          vertex.setValue(zombieText);

          LongWritable newMessage = new LongWritable();
          newMessage.set(superstep+1);

          aggregate("zombie.count",longIncrement );

          for (Edge<LongWritable, LongWritable> edge : vertex.getEdges()) {
            this.sendMessage(edge.getTargetVertexId(), newMessage);
          }
        }
      } else {
        if (vertex.getValue().toString().equals("Human")) {

          Iterator<LongWritable> it = messages.iterator();
          if (it.hasNext()) {
            zombieText.set("Zombie." + superstep);
            vertex.setValue(zombieText);
            aggregate("zombie.count",longIncrement );

            LongWritable newMessage = new LongWritable();
            newMessage.set(superstep+1);

            for (Edge<LongWritable, LongWritable> edge : vertex.getEdges()) {
              this.sendMessage(edge.getTargetVertexId(), newMessage);
            }
          } else {
            vertex.voteToHalt();
          }

        } else {
          vertex.voteToHalt();
        }
      }
    }
  }
```

本代码中存在许多具体的逻辑，我们会深入地研究每个操作，但是首先要明白 compute() 方法如何调用。每个点均会调用 compute() 方法，而且在最后一个 superstep 结束时，所有消息的迭代器会将消息发送到点上。

逻辑流程如下所示。

- 如果这是第一个 superstep，而且我是一个僵尸，那么我周围的每一个人都要被咬。
- 如果在第一个 superstep 之后，而且我是一个接受咬人消息的人，那么我自己会变成僵尸，还会去咬我周围的每一个人。
- 如果我是一个僵尸，而且被咬了，那么不会有任何改变。

同时需要注意的是，我们提出在两个位置停止：僵尸被咬或者人类未被咬。这样做的原因在于有以下两种最终状态。

- 我们可以让每个人都被咬然后变成僵尸，而在那一点上我们需要停止数据处理。
- 我们可以使僵尸与人类之间没有直接相连的边。所以这些人类永远都不会变成僵尸。

5.4.3 将图回写磁盘

图中已经产生了很多僵尸，现在应该将结果写回磁盘了。我们使用 VertexOutputFormat 完成写回数据。这里不会涉及细节内容，只要注意这与 InputFormat 相反即可。

```
public class ZombieTextVertexOutputFormat
  extends TextVertexOutputFormat<LongWritable, Text, LongWritable> {

  @Override
  public TextVertexWriter createVertexWriter(TaskAttemptContext context)
    throws IOException, InterruptedException {
    return new ZombieRecordTextWriter();
  }

  public class ZombieRecordTextWriter extends TextVertexWriter {
    Text newKey = new Text();
    Text newValue = new Text();

    public void writeVertex(Vertex<LongWritable, Text, LongWritable> vertex)
      throws IOException, InterruptedException {
      Iterable<Edge<LongWritable, LongWritable>> edges = vertex.getEdges();

      StringBuilder strBuilder = new StringBuilder();

      boolean isFirst = true;
      for (Edge<LongWritable, LongWritable> edge : edges) {
        if (isFirst) {
          isFirst = false;
        } else {
          strBuilder.append(",");
        }
        strBuilder.append(edge.getValue());
      }

      newKey.set(vertex.getId().get() + "|" + vertex.getValue() + "|"
                 + strBuilder.toString());

      getRecordWriter().write(newKey, newValue);
    }
```

```
    }
  }
```

5.4.4 整体流程控制

现在，与 MapReduce 中的情况类似，我们需要将一切都设定好，而且在主函数中进行配置。代码如下。

```
public class ZombieBiteJob implements Tool {
  private static final Logger LOG = Logger.getLogger(ZombieBiteJob.class);
  private Configuration conf;

  @Override
  public void setConf(Configuration conf) {
    this.conf = conf;
  }

  @Override
  public Configuration getConf() {
    return conf;
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 3) {
      throw new IllegalArgumentException(
          "Syntax error: Must have 3 arguments " +
          " <numbersOfWorkers> <inputLocaiton> <outputLocation>");
    }

    int numberOfWorkers = Integer.parseInt(args[0]);
    String inputLocation = args[1];
    String outputLocation = args[2];

    GiraphJob job = new GiraphJob(getConf(), getClass().getName());
    GiraphConfiguration gconf = job.getConfiguration();
    gconf.setWorkerConfiguration(numberOfWorkers, numberOfWorkers, 100.0f);

    GiraphFileInputFormat.addVertexInputPath(gconf,
                                             new Path(inputLocation));
    FileOutputFormat.setOutputPath(job.getInternalJob(),
                         new Path(outputLocation));

    gconf.setComputationClass(ZombieComputation.class);
    gconf.setMasterComputeClass(ZombieMasterCompute.class);
    gconf.setVertexInputFormatClass(ZombieTextVertexInputFormat.class);
    gconf.setVertexOutputFormatClass(ZombieTextVertexOutputFormat.class);
    gconf.setWorkerContextClass(ZombieWorkerContext.class);

    boolean verbose = true;
    if (job.run(verbose)) {
      return 0;
    } else {
      return -1;
```

```
      }
    }

    public static void main(String[] args) throws Exception {
      int ret = ToolRunner.run(new ZombieBiteJob(), args);
      if (ret == 0) {
        System.out.println("Ended Good");
      } else {
        System.out.println("Ended with Failure");
      }
      System.exit(ret);
    }
  }
```

5.4.5 何时选用Giraph

Giraph 的功能非常强大，但是就像我们从概念和需要的代码中看到的，它会挑战你的心理承受能力。不过，如果需要进行图数据处理，保证确切的服务等级协议（Service-Level Agreement，SLA），而且需要成熟的解决方案，那么可以选择 Giraph。

但是要注意，目前并不是所有主流 Hadoop 厂商发行的版本都支持 Giraph。这并不是说 Giraph 不能用于你所选择的发行版本，只是可能需要与 Hadoop 厂商进一步沟通，看一下是否能获得这类支持。你至少可以为自己的发行版本编译 Giraph 对应的 Jar 包。

5.5 GraphX

第 4 章从替代 MapReduce 进行 ETL 的角度，对 Spark 进行了大量讨论。但是对于图数据处理，Spark 是怎样成为 MapReduce 有力竞争者的呢？ Spark 的 GraphX 解决了 MapReduce 的两个主要问题，而 Pregel 与 Giraph 也是为了解决这两个问题而设计的：对数据迭代处理进行提速，并让数据尽可能接近内存（在内存中进行处理）。Spark 的 GraphX 作为图数据处理框架也提供了另一个优势，即图只是另外一种类型的 RDD。也就是说，对于已经了解 Spark 的开发者来说，GraphX 是一个熟悉的编程模型。

公平地说，Spark 上的图数据处理仍然很新，这也反映在我们看到的代码案例中。你会注意到，它是使用 Scala 语言实现的。这是因为在本文写作时还没有可用的 Java API。

5.5.1 另一种RDD

Spark 中存在几种基本的 RDD。其中最常见的是 RDD 与 PairRDD。你可能已经猜到了，要从常用的 Spark 中进行图数据处理，只需要创建一个 EdgeRDD 和一个 VertexRDD，两者都是常见 RDD 对象的简单扩展。

我们能够通过一个简单的 Map 转化功能，从任何一个 RDD 得到 EdgeRDD 或 VertexRDD。这并不神奇。从普通 Spark 到 GraphX 的最后一步是将两个新的 RDD 放置到一个 `Graph` 对

象中。该 Graph 对象只包含这两个 RDD 的引用，而且提供对其进行图处理的方法。

图处理模式很容易通过 GraphX 进入，也很容易由此脱离。任何时候我们都能够从 Graph 对象中输出 EdgeRDD 或 VertexRDD，并且开始对其执行一般的 RDD 功能。

我们来看一下代码，这样更容易理解。以下代码得到了一个 SparkContext，而且创建了得到一个图所需要的两个 RDD。

```
val sc =
    new SparkContext(args(0), args(1), args(2), Seq("GraphXExample.jar"))

//为点创建RDD
val users: RDD[(VertexId, (String))] =
  sc.parallelize(Array(
      (1L, ("Human")),
      (2L, ("Human")),
      (3L, ("Human")),
      (4L, ("Human")),
      (5L, ("Human")),
      (6L, ("Zombie")),
      (7L, ("Human")),
      (8L, ("Human")),
      (9L, ("Human")),
      (10L, ("Human")),
      (11L, ("Human")),
      (12L, ("Human")),
      (13L, ("Human")),
      (14L, ("Human")),
      (15L, ("Human")),
      (16L, ("Zombie")),
      (17L, ("Human")),
      (18L, ("Human")),
      (19L, ("Human")),
      (20L, ("Human")),
      (21L, ("Human")),
      (22L, ("Human")),
      (23L, ("Human")),
      (24L, ("Human")),
      (25L, ("Human"))
      ))
//为边创建RDD

val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(
      Edge(10L, 9L, "X"),
      Edge(10L, 8L, "X"),
      Edge(8L, 7L, "X"),
      Edge(8L, 5L, "X"),
      Edge(5L, 4L, "X"),
      Edge(5L, 3L, "X"),
      Edge(5L, 2L, "X"),
      Edge(9L, 6L, "X"),
      Edge(6L, 1L, "X"),
      Edge(20L, 11L, "X"),
```

```
        Edge(20L, 12L, "X"),
        Edge(20L, 13L, "X"),
        Edge(5L, 13L, "X"),
        Edge(20L, 14L, "X"),
        Edge(11L, 15L, "X"),
        Edge(20L, 16L, "X"),
        Edge(14L, 17L, "X"),
        Edge(1L, 17L, "X"),
        Edge(20L, 18L, "X"),
        Edge(21L, 18L, "X"),
        Edge(21L, 22L, "X"),
        Edge(4L, 23L, "X"),
        Edge(25L, 15L, "X"),
        Edge(24L, 3L, "X"),
        Edge(21L, 19L, "X")
        ))

    //创建一个默认用户,以防万一有关系丢失对应的用户

    val defaultUser = ("Rock")
    //创建初始的图
  val graph = Graph(users, relationships, defaultUser)
  graph.triangleCount
```

这里发生的事情同样谈不上神奇。SparkContext 的 parallelize() 方法与我们用来创建正常 RDD 的方法相同。注意，我们在代码中填充了相同的数据，所以读起来更容易理解。但是数据也很容易从磁盘中加载。

在介绍数据处理代码之前，我们先花一分钟的时间来讨论一下 defaultUser 变量。在 GraphX 中，你可以选择针对点进行打捞（catch-all vertex），这就是所谓的 default。如果发送到一个不存在的点，那么消息就会发送到 default 上。在这个例子中，如果一个僵尸尝试咬一个不存在的人，那么这个僵尸就会很不幸地被绊倒然后咬到石头。但是不用担心，僵尸已经死了，所以不会受伤。

5.5.2　GraphX的Pregel接口

在研究代码之前我们需要考虑两个因素。首先，GraphX Pregel API 与 Giraph 不同。第二，代码在 Scala 中，所以非常紧凑。因此我们需要花费一些时间将其分解。虽然如此，但是仍然需要注意，我们在 Giraph 中写的 300 多行代码现在已经通过 GraphX 被 Scala 写的 20 行代码所替换。

```
  //所有的僵尸咬人的逻辑在此
  val graphBites = graph.pregel(0L)(
    (id, dist, message) => {
      if (dist.equals("Zombie")) {
        (dist + "_" + message)
      } else if (message != 0){
        "Zombie" + "_" + message
      } else {
        dist + "|" + message
      }
```

```
    }, triplet => {
      if (triplet.srcAttr.startsWith("Zombie") &&
          triplet.dstAttr.startsWith("Human")) {
        var stringBitStep =
            triplet.srcAttr.substring(triplet.srcAttr.indexOf("_") + 1)
        var lastBitStep = stringBitStep.toLong
        Iterator((triplet.dstId, lastBitStep + 1))
      } else if (triplet.srcAttr.startsWith("Human") &&
                 triplet.dstAttr.startsWith("Zombie")) {
        var stringBitStep =
            triplet.dstAttr.substring(triplet.dstAttr.indexOf("_") + 1)
        var lastBitStep = stringBitStep.toLong
        Iterator((triplet.srcId, lastBitStep + 1))
      } else {
        Iterator.empty
      }
    },
    (a, b) => math.min(b, a))

graphBites.vertices.take(30)
```

如果能自己解耦这个代码，那么你可能比作者更渊博。我们先定义 `graph.pregel()` 方法，分解代码。

`graph.pregel()` 包含两个参数：第一个是值的集，第二个是函数集，二者将同时执行。

在第一个参数集当中，我们只使用第一个参数，定义如下。

- 第一条消息。
- 最大迭代次数（使用默认值即可，不使用该参数）。
- 边的方向（使用默认值即可，不使用该参数）。

我们发送的第一条消息为 0。注意，与 Giraph 不同，我们不能访问一个 `superstep` 的数值。所以对于该例，我们会追踪消息中的 `superstep`。这并不是执行 `superstep` 的唯一方法。我们随后会深入讨论这部分内容。

如果没有更多消息发送，而且我们想继续进行直到达到那个点，那么 GraphX 会停止运行，所以我们没有设置最大迭代。

第二个参数集包含一系列数据处理方法。

- `vprog()`

  一种用户自定义的点程序。换句话说，当一条消息到达一个点时它能够确定如何处理。

- `sendMessage()`

  该方法包含点发送信息的逻辑（可能要发送信息，可能不要发送信息）。

- `mergeMessage()`

  该方法包含一种功能，能确定到达同一个点的消息如何合并。当数百万个点将一条信息发送到同一个点时，该功能会非常实用。

5.5.3 vprog()

现在，我们一起深入讨论一下 GraphX 代码中的每一种方法，首先是 vprog() 方法。

```
(id, dist, message) => {
  if (dist.equals("Zombie")) {
    (dist + "_" + message)
  } else if (message != 0){
      "Zombie" + "_" + message
  } else {
      dist + "|" + message
  }
}
```

我们可以看到这种方法获取了点的 ID、包含点的信息的对象以及被发送到点的消息。在这种方法中，我们确定了是否需要修改点的信息对象，以对即将达到的消息做出反应。

该代码中的逻辑非常简单。如果消息是 0（即第一个 superstep），则更新僵尸，将其设置为 Zombie_0。如果消息大于 0，则将人改为一个 Zombie_{messageNUM}。否则不需要修改点。

5.5.4 sendMessage()

下一种方法规定了发送消息的时间。有趣的是，GraphX 与 Giraph 相反，在计划将消息发出时能够访问目标点的信息。这样咬人信息就不会发送到其他僵尸对应的点上，咬人的消息不会浪费。

在这种方法中你可以看到，我们能够访问 triplet 对象。该对象包含了我们需要的源、目标点以及相连接的边相关的所有信息。

```
triplet => {
   if (triplet.srcAttr.startsWith("Zombie") &&
       triplet.dstAttr.startsWith("Human")) {
     var stringBitStep =
         triplet.srcAttr.substring(triplet.srcAttr.indexOf("_") + 1)
     var lastBitStep = stringBitStep.toLong
     Iterator((triplet.dstId, lastBitStep + 1))
  } else if (triplet.srcAttr.startsWith("Human") &&
             triplet.dstAttr.startsWith("Zombie")) {
    var stringBitStep =
        triplet.dstAttr.substring(triplet.dstAttr.indexOf("_") + 1)
    var lastBitStep = stringBitStep.toLong
    Iterator((triplet.srcId, lastBitStep + 1))
  } else {
    Iterator.empty
  }
}
```

此处的逻辑非常简单：如果源是一个僵尸，而且目标是人类，那么发送一个咬人的消息。

5.5.5 mergeMessage()

最后一个方法是目前来说最简单的方法。我们需要做的是合并消息，所以我们只需要咬一次人。多次咬人并不会制造出更多僵尸，而只会浪费网络带宽。

```
(a, b) => math.min(b, a)
```

本例执行了 `math.min(b, a)`，但是这并不是必需的。原因在于 `b` 与 `a` 通常是同一个值。我们使用 `math.min()` 向你展示该项功能中能够使用的方法。下面的代码也能执行这种操作，但是可能没那么容易理解。

```
(a, b) => a
```

GraphX 与 Giraph

在比较 GraphX 与 Giraph 时你会注意到，GraphX 中丢失了很多东西，包括下面这些。

- 主线程计算
- 阻塞等待
- 工作线程计算
- 超步

第一次接触 GraphX 时，你可能会很沮丧地发现 Giraph 中所有工具都不见了，直到你意识到 GraphX 是一种基于 Spark 的图处理。那么，这又意味着什么呢？

记住 Giraph 的三个阶段：读取、处理与写入。但是对于开发者来说，GraphX 中的这三个阶段并不是很明确，因为开发者可以通过 Spark 按照需求随意混合处理类型。因此，应用能够执行从 Spark 转化到 GraphX 的处理，而后返回到 Spark。这一功能使得你能够在应用中混合处理模型。于是，GraphX 中三个阶段并不是很明确，但是 Spark 模型为数据处理提供了更大的灵活性。

5.6　工具选择

在本书写作时，从成熟度方面来讲，Giraph 优于 GraphX，但是随着时间的演进，这个差距会缩小。短期之内，你可以这样进行选择：如果要处理的全都是图数据，而且需要可伸缩的最稳定方法，那么你最好选择 Giraph。如果图数据处理只是解决方案中的一环，而且集成与代码的整体灵活性比较重要，那么 GraphX 可能是一个不错的选择。

5.7　小结

长期以来，Graph 在计算机科学中都是一个核心概念。通过集成图处理与 Hadoop，我们现在能够有效地应对极大的图数据，而且花费的时间比以前的系统少。

尽管 Hadoop 中的图数据处理生态系统相对来说仍然很年轻，但是它发展得很快，而且极其适合解决各种类型的问题（如人、地点或者事物之间的关系分析）。当这些关系方面的数据持续激增时，存储与数据处理能力也需要随之增加，这个时候 Hadoop 中的图数据处理就更有优势了。

本章旨在提供一种思路，让你了解什么样的应用适合采用图数据处理。另外，本章概述了目前用于图数据处理的 Hadoop 工具，以指导你选择正确的工具。

第 6 章

协调调度

工作流协调调度（workflow orchestration），也称工作流自动化（workflow automation）或者业务处理自动化（business process automation），实际上指计划、调度以及管理工作流的任务。工作流（workflow）是一系列的数据处理操作。能够执行自动化流程的系统称作协调调度框架（orchestration framework）或者工作流自动化框架（workflow automation framework）。

工作流协调调度是应用框架中很重要的一部分，但是经常被忽略。它在 Hadoop 中尤为重要，原因在于很多应用创建时都是一个个的 MapReduce 任务，而在一个单一任务中能够执行的操作非常有限。即使 Hadoop 工具集不断地发展，类似于 Spark 的更加灵活的数据处理框架逐渐占据主导地位，将复杂的数据处理工作流分解，输入可以重复使用的组件中，以及使用外部引擎处理将结果再聚合到一起，也还是很有益处的。

6.1　工作流协调调度的必要性

使用 Hadoop 创建端到端（end-to-end）的应用通常涉及一些数据处理步骤。用户可能会通过 Sqoop 获取关系型数据库中的数据，并将其输出至 Hadoop，然后运行 MapReduce 任务，根据一些数据约束校验数据，将数据转化为更适合的格式。然后，用户可能会执行一些 Hive 任务，聚合并分析数据。特别关注分析时，可能需要增加额外的 MapReduce 步骤。

每一个任务都可以看作一个操作（action）。这些操作需要调度、协调与管理。

比如，用户可能会这样调度一个操作。

- 操作在一个特定时间执行。
- 操作在一个周期性的间隔之后执行。
- 操作在一个事件发生时（如，当文件可用时）执行。

用户可能会想这样协调一个操作。

- 前一个操作成功完成时，开始运行这个操作。

用户可能想这样管理一个操作。

- 操作成功或失败，需要发送电子邮件通知。
- 记录完成操作所需要的时间，或者完成操作的不同步骤所需要的时间。

以上一系列的操作或者工作流可以通过一个有向无环图（Directed Acyclic Graph，DAG）来表示。执行工作流时，操作要么并行，要么根据前次操作的结果执行。

工作流协调调度将复杂的处理过程分解形成简单的、可重复使用的单元，这正是良好的工程实践。工作流引擎有助于确定工作流组件之间的接口。另外，这里不需要考虑应用的逻辑，因为使用的是自动化流程系统已经存在的一部分。

良好的工作流自动化流程引擎能够支持这样一些业务需求：元数据与数据血缘跟踪（追踪特定数据如何通过转化和合并步骤进行修改）、企业中不同软件之间的集成、数据生命周期管理以及数据质量的追踪与报告。这里也支持运维功能，如管理工作流组件的存储库、规划灵活的工作流、管理依赖关系、监控来自集中化位置的工作流状态、重试失败的工作流、生成报表以及监测到问题出现时回滚工作流。

下面我们来讨论一下对于架构师来说，有哪些基于 Hadoop 的应用进行协调调度的常见方法。

6.2 脚本的局限性

第一次部署基于 Hadoop 的应用时，你可能倾向于使用某种脚本语言（如 Bash、Perl 或 Python）将不同的操作结合到一起。如果工作流比较短而且不太重要，那么这种方法是可行的，而且实施起来通常比其他方法更简单。

Bash 脚本通常如例 6-1 所示。

例 6-1 workflow.sh

```
sqoop job -exec import_data
if beeline -u "jdbc:hive2://host2:10000/db" -f add_partition.hql 2>&1 | grep FAIL
then
        echo "Failed to add partition. Cannot continue."
fi

if beeline -u "jdbc:hive2://host2:10000/db" -f aggregate.hql 2>&1 | grep FAIL
then
        echo "Failed to aggregate. Cannot continue."
fi
sqoop job -exec export_data
```

而且用户会经常执行该脚本，时间为每天上午 1 点，使用的 crontab 项如例 6-2 中所示。

例 6-2 crontab 项

```
0 1 * * * /path/to/my/workflow.sh > /var/log/workflow-`date +%y-%m-%d`.log
```

当该脚本投入生产，有人依赖脚本的结果时，就会有其他需求出现。你可能想更好地处理错误、通知用户以及其他监控工作流状态的系统、追踪工作流（不管是作为整体的工作流还是某个单一操作）的执行时间、更精细化地处理工作流不同步骤之间的逻辑、重新运行整个工作流或者部分工作流，以及在不同工作流中复用已有的组件。

在产品需求不断升级的情况下，维持一个内部自动化脚本会很困难，相当于从零开始。尽管有很多工具具有这样的功能，我们还是建议熟悉其中的一个，以此满足自动化流程的需求。

需要明白的是，迁移一个现存的工作流脚本使之能够在一个工作流管理器中运行，这种操作一般来说并不是很复杂。如果各步骤之间相对独立——如例 6-1，每个步骤都使用前一个步骤写入 HDFS 中的文件——那么转化会比较简单。不过，如果脚本使用标准输出或者本地文件系统在不同步骤之间传递信息，那么这个过程可能会很复杂。如果后期有迁移计划，那么在选择使用脚本中创建工作流，或是在工作流管理器中创建工作流的时候，就需要慎重考虑以上因素。

6.3　企业级任务调度器及Hadoop

很多公司的工作流自动化和调度软件都有一个内部统一的标准框架。常见的选择包括 ControlM、UC4、Autosys 与 ActiveBatch。使用现存的软件调度 Hadoop 工作流是一个比较合理的选择。这样可以保证复用现有基础设施，而且不需要学习其他的框架。不仅是针对 Hadoop，将多种系统集成到同一工作流时，这种方法总会使集成变得更轻松。

一般来说，这些系统会在每个服务器上安装 agent，以便在每个服务器中执行各种操作。在 Hadoop 集群中，用户的工具与应用 JAR 包通常在边缘节点（edge node，也称 gateway node，网关节点）上部署。使用这些工具设计工作流时，开发者通常会指定执行操作的服务器，然后指定 agent 在服务器上执行的查询命令。例 6-1 中的命令分别为 `sqoop` 与 `beeline`。agent 将执行命令，等待命令的完成并且将返回状态报告给工作流管理器。设计工作流时，开发者能够指定规则，即如何处理每个任务以及整个工作流的成功或者失败。同一个企业作业调度程序也能用于调度工作流，使其在指定的时间或者按照指定间隔周期性运行。

> 这些企业工作流自动化系统并不是针对 Hadoop 而创建的，所以使用这些系统的详细概述不在本书的讨论范围之内。我们将重点介绍属于 Hadoop 生态系统的框架。

6.4　Hadoop生态系统中的工作流框架

Hadoop 生态系统中有一些工作流引擎。这些引擎紧密集成并且拥有内部支持。因此，对于很多企业来说，如果需要调度 Hadoop 工作流，却没有标准的自动化解决方法，那么可以选择其中一个工作流引擎用于工作流自动化与调度。

对于分布式系统，较为流行的开源工作流包括 Apache Oozie、Azkaban、Luigi 与 Chronos。

- Oozie（http://oozie.apache.org/）由 Yahoo! 创建，旨在支持其逐渐发展的 Hadoop 集群以及在集群上不断增加的任务与工作流。
- Azkaban（http://azkaban.github.io/）由 LinkedIn 创建，目的是对工作流进行可视化的便捷管理。
- Luigi（https://github.com/spotify/luigi）是一种来自 Spotify 的开源 Python 包。Spotify 能够帮助用户自动化调度长期运行的批任务，而且内置支持 Hadoop。
- Chronos（http://mesos.github.io/chronos）是一种来自 Airbnb 的开源分布式调度器，能够容错，在 Mesos 框架上运行。这是一种用于替代 cron 定时任务的分布式系统。

本章将重点放在 Oozie 上，展示如何使用 Oozie 创建工作流。选择 Oozie 的原因是每一个 Hadoop 发行版都包含它。其他自动化流程引擎的作用相似，但是在语法与细节上有所不同。

选择工作流引擎时，需要考虑以下因素。

- *安装简便程度*

  工作流引擎是否足够容易安装？版本升级会不会很麻烦？

- *社区参与和认同*

  在生态系统中，社区是否能够快速对有前景的新项目增加支持？当新的项目增加到生态系统中时（如，Spark 是目前相当流行的新增项目），用户希望工作流引擎能够支持新的项目，这样不会阻碍用户在工作流中使用更新的项目。

- *用户接口支持*

  将工作流创建为文件还是通过 UI 访问？如果是文件，创建和更新这些文件是否容易？如果是 UI 方式，UI 有多强大、多直观？

- *测试*

  在工作流形成后，如何测试它们？

- *日志*

  引擎是否提供方便的日志访问方式？

- *工作流管理*

  引擎提供的管理水平符合你的期望吗？引擎能够跟踪全部工作流或部分工作流所花费的时间吗？能够灵活控制操作的 DAG（比如基于前一次的操作输出做出决策）吗？

- *错误处理*

  如果出现错误，你能通过引擎重新运行任务或任务的一部分吗？你能通知用户吗？

6.5 Oozie术语

深入讨论之前，我们先看一下 Oozie 术语。

- *工作流操作*

  自动化流程引擎能够执行的单一任务单元（如，一次 Hive 查询、一个 MapReduce 任

务、一次 Sqoop 输出）。

- *工作流*
  一个操作（或任务）的控制依赖 DAG。
- *协调器*
  数据集与数据表的定义，能够触发工作流。
- bundle
  协调器的集合。

6.6 Oozie概述

Oozie 可以说是 Hadoop 上最受欢迎的可伸缩分布式工作流引擎。至少大部分 Hadoop 发行版都装载了 Oozie。Oozie 的主要优势是能够与 Hadoop 生态系统深入集成，而且能够处理数千个并发的工作流。

对于常用的 Hadoop 生态系统组件（如 MapReduce、Hive、Sqoop 与 distcp），Oozie 含有多个内置的操作。因此，Oozie 易于创建这些组件的工作流。另外，Oozie 还能执行任意一个 Java 程序与 shell 脚本。

我们在这里简单地介绍一下 Oozie，然后提出使用 Oozie 所需要考虑的因素与建议。

Oozie 的主要逻辑组件如下所述。

- *一个工作流引擎*
  用于执行工作流，包括各种操作，如 Sqoop、Hive、Pig 与 Java。
- *一个调度器（即所谓的协调器）*
  基于频率或者前一个数据集位置中已存在的数据集调度工作流。
- REST API
  包括 API，用来执行、调度与监控工作流。
- *命令行客户端*
  调用 REST API，并且能够让用户通过命令行执行、调度及监控任务。
- bundle
  即协调器应用的集合，能够统一控制。
- *提醒*
  当任务状态发生改变时（当任务开始、结束、出现错误或者转移到下一个操作，等等），将事件发送到外部 JMS 队列。也支持外部应用与工具之间的简单集成。
- SLA *监控*
  根据开始时间、结束时间或者持续时间追踪任务的 SLA。当任务符合 SLA，或无法保证 SLA 时，Oozie 会通过 Web 指示板、REST API、JMS 队列或者 Email 通知用户。

- *后端数据库*

  存储 Oozie 的持久信息，包括协调器、bundle、SLA 与工作流的历史信息。该数据库可以是 MySQL、Postgres、Oracle 或者 MSSQL。

Oozie 以客户端—服务器模型工作。图 6-1 展示了 Oozie 的客户端—服务器架构。

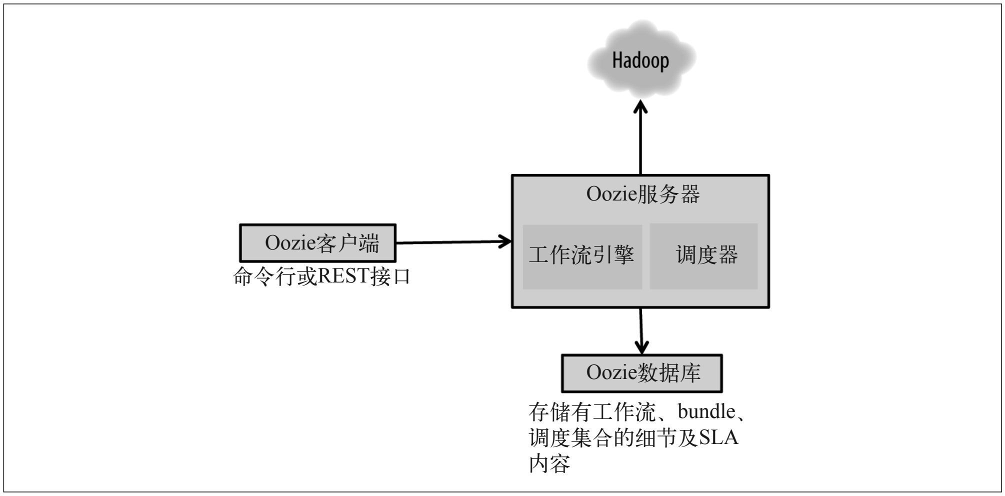


图 6-1：Oozie 架构

执行一个工作流时，事件出现的顺序如图 6-2 所示。

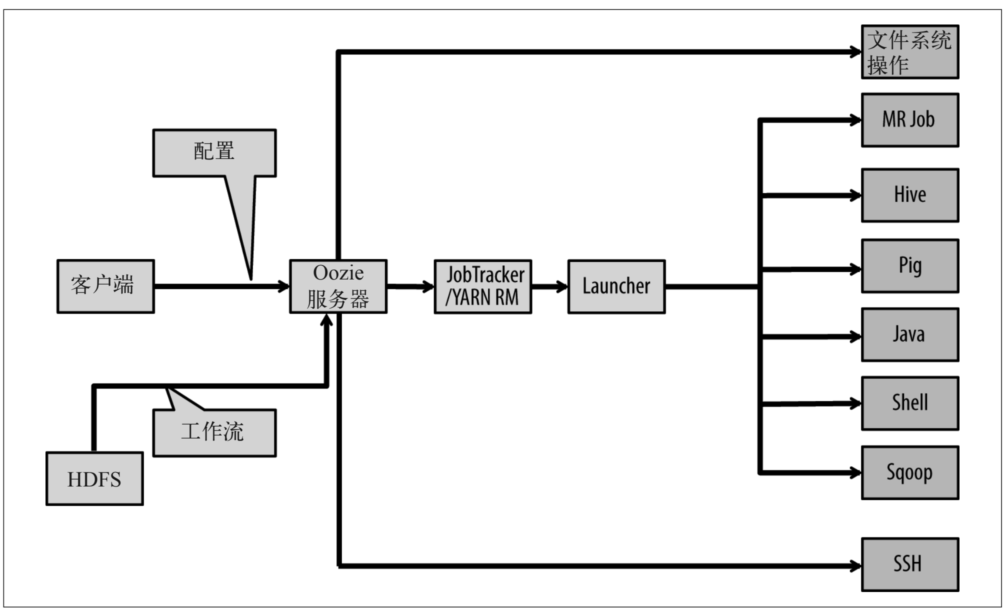


图 6-2：执行工作流时 Oozie 中事件出现的顺序

如图 6-2 所示，客户端与 Oozie 服务器相连接，而且提交任务配置。一系列的键值对确定了任务执行的重要参数，而不是工作流本身进行确定。工作流包含一系列操作以及连接这些操作的逻辑，由一个名为 workflow.xml 的文件定义。任务配置必须包括 workflow.xml 文件在 HDFS 上的位置，也能收录其他参数，通常包括 NameNode、JobTracker（如果使用 MapReduce v1 [MR1]）或者 YARN Resource Manager（如果使用 MapReduce v2 [MR2]）的 URI 地址。这些参数随后会用于定义工作流。Oozie 服务器接收客户端提供的配置时，会读取 workflow.xml 文件与来自 HDFS 的工作流定义。随后，Oozie 服务器解析工作流定义，并启动工作流文件所述的各个操作。

为什么 Oozie 服务器会使用一个启动器

Oozie 服务器不会直接执行操作。对于任何操作，Oozie 服务器只通过一个 Map 任务（被称为 launcher 任务）来启动 MapReduce 任务，然后启动 Pig 或 Hive 操作。这一架构上的设计使 Oozie 服务器更轻量、可伸缩性更好。

如果所有的操作都由一个单一的 Oozie 服务器启动、监控及管理，那么所有的客户端库都必须保存在这个服务器上。因为所有的操作都通过同一个节点执行，所以该服务器的负担会变得很重，可能会成为瓶颈。但是，单一的启动器任务能够使 Oozie 使用已存的分布式 MapReduce 框架，委托启动器任务启动、监控与管理操作，进而委托启动器任务运行的节点执行这些操作。该启动器会从 Oozie 的 sharedlib（HDFS 上的一个目录，包含不同 Oozie 操作所要求的所有客户端库）中提取所需的客户端库。上述的例外情况是文件系统、SSH 以及 Email 相关的操作，Oozie 服务器直接执行这些操作。

比如，运行一个 Sqoop 操作，Oozie 服务器将启动一个 mapper MapReduce 任务，称作 sqoop-launcher。该 mapper 将运行 Sqoop 客户端，而后依次运行其自身的 MapReduce 任务（称作 sqoop-action）。执行该任务配置的 mapper 数量与 Sqoop 的相同。对于 Java 或 shell 操作，启动器将执行 Java 或 shell 应用，而且不会产生其他的 MapReduce 任务。

6.7 Oozie工作流

如前所述，工作流在 Oozie 中的定义被写入了 XML 文件，文件名称为 workflow.xml。一个工作流包含操作节点和控制节点。操作节点负责运行实际操作，而控制节点控制执行的流程。控制节点能够启动或结束一个工作流、制定决策、拆分或合并执行操作，或者终止执行任务。workflow.xml 文件代表控制节点与操作节点的 DAG，用 XML 表述。关于 XML 文件模式描述的完整讨论不在本书范围之内，你可以参阅 Apache Oozie 官方文档（https://oozie.apache.org/docs/4.0.1/WorkflowFunctionalSpec.html#OozieWFSchema）。

> Oozie 的默认网络接口使用 ExtJS。但是，对于开源 UI 工具 Hue（http://www.gethue.com，Hadoop 用户的体验），Oozie 应用通常使用 UI 创建或设计 Oozie 工作流。Hue 基于 UI 设计的工作流产生 workflow.xml 文件。

Azkaban

Azkaban 是一种开源自动化流程引擎，来自 LinkedIn。与 Oozie 相同，在 Azkaban 中也能编写工作流并且调度工作流（简称 flow）。Azkaban 与 Oozie 的创建初衷不同：Oozie 的目标是在一个非常大的集群上管理数千个任务，因此重点在于可伸缩性，Azkaban 的主要作用与重点在于简洁和可视化。

Azkaban 提供了两项服务与一个后端数据库。服务包括 Azkaban Web 服务器与 Azkaban 执行器。每一个服务项目都能在不同的主机上运行。Azkaban Web 服务器不仅是一种 Web UI，也是调度所有 flow 的主要控制器，通过 Azkaban 运行。它主要负责管理、调度与监控。Azkaban 执行器负责在 Azkaban 中执行 flow。目前 Azkaban Web 服务器是一个单一故障点，同时支持多个 Azkaban 执行器，提供高可用性和可伸缩性。

Azkaban Web 服务器与 Azkaban 执行器都能与后端的 MySQL 数据库会话，能够存储工作流集合、工作流的调度、SLA、工作流的执行状态以及工作流的历史。

Azkaban 内部只能支持运行本地 UNIX 命令与简单的 Java 项目。但是，Azkaban 任务类型的插件很容易安装，支持 Hadoop、Hive 与 Pig 任务。Azkaban Web 服务器包括很多其他有用的插件：用于浏览 HDFS 目录的 HDFS Browser（与 Hue 的 HDFS Browser 相似）、通过安全途径与 Hadoop 集群会话的 Security Manager、为任务运行提供总结的 Job Summary、将 Pig 任务可视化的 Pig Visualizer，以及用于创建、管理与运行报告的 Reportal。

图 6-3 为 Azkaban 的架构。

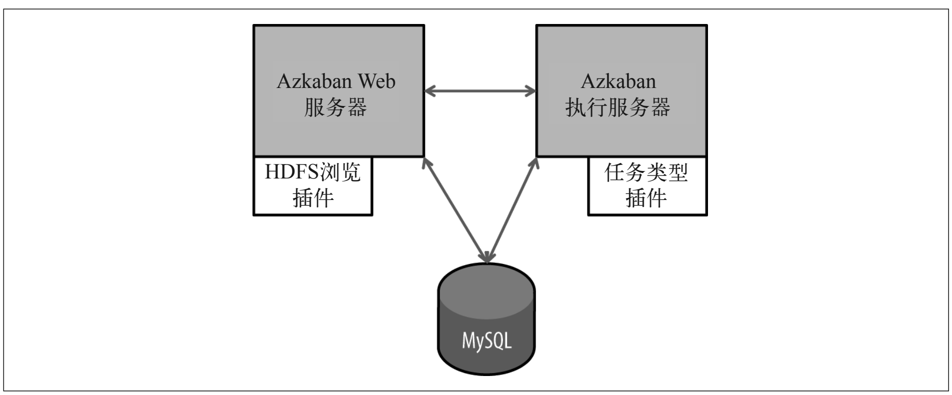


图 6-3：Azkaban 架构

表 6-1 对于 Oozie 和 Azkaban 的比较不算详细，但是列出了它们之间的差异。

表6-1：比较Oozie与Azkaban

标准	Oozie	Azkaban
工作流定义	使用 XML，需要加载到 HDFS 中	使用简单的声明 .job 文件。需要通过网络接口安装并加载
可伸缩性	生成一个单一的、仅限于 Map 的任务，称为 Launcher，用于管理启动的任务。系统的可伸缩性更好，但是在调试时增加了额外的间接层次	通过一个单一的执行器生成所有的任务，可能会引发伸缩性相关的问题
Hadoop 生态系统的支持	提供了很好的支持。能够运行 MapReduce、Streaming MapReduce、Pig、Hive、Sqoop 与 Distcp actions	支持 Java MapReduce、Pig、Hive 与 VoldemortBuildAndPush（能够将数据输入 Volde mort 键值存储中）任务。其他任务需要按照命令任务类型实施
安全性	与 Kerberos 集成	支持 Kerberos，但在安全的 Hadoop 集群上只支持旧版本的 MR1 API 执行 MapReduce 任务
工作流版本管理	通过 HDFS 符号链接提供简单的工作流版本化	通过网络 UI 加载，提供版本化工作流的明确方法
与 Hadoop 生态系统之外的系统集成	支持 HDFS、S3 或者本地文件系统上的工作流（尽管不推荐本地文件系统）。支持邮件、SSH、shell 等操作。提供编写自定义操作的 API	未要求工作流存储于 HDFS 上；可能可以完全用于 Hadoop 之外的系统
工作流的参数化	支持通过变量与 Expression Language（EL）函数对工作流进行参数化（如 `${wf:user()}`）	工作流只能通过变量进行参数化
工作流调度	支持时间与数据触发	只支持时间触发
与服务器之间的交互	支持 REST API、Java API 与 CLI，拥有网络接口（内置、通过 Hue）	拥有较好的网络接口，能够对网络服务器触发工作流发送 cURL 命令，无正式的 Java API 或 CLI

6.8 工作流范式

认识了常用的自动化流程引擎，下面我们看一些行业中常见的工作流范式。

6.8.1 点对点式工作流

按顺序执行操作时经常会用到这种类型的工作流，比如，如果你想使用 Hive 执行数据集的合并操作，而且在成功后使用 Sqoop 将其输入到 RDBMS 中。

在 Oozie 中实施时，workflow.xml 文件如下所示：

```
<workflow-app xmlns="uri:oozie:workflow:0.4" name="aggregate_and_load">
   <global>
         <job-tracker>${jobTracker}</job-tracker>
```

```
        <name-node>${nameNode}</name-node>
    </global>

    <start to="aggregate" />

    <action name="aggregate">
      <hive xmlns="uri:oozie:hive-action:0.5">
        <job-xml>hive-site.xml</job-xml>
        <script>populate_agg_table.sql</script>
      </hive>
      <ok to="sqoop-export" />
      <error to="kill" />
    </action>

    <action name="sqoop_export">
      <sqoop xmlns="uri:oozie:sqoop-action:0.4">
        <arg>export</arg>
        <arg>--connect</arg>
        <arg>jdbc:oracle:thin:@//orahost:1521/oracle</arg>
        <arg>--username</arg>
        <arg>scott</arg>
        <arg>--password</arg>
        <arg>tiger</arg>
        <arg>--table</arg>
        <arg>mytable</arg>
        <arg>--export-dir</arg>
        <arg>/etl/BI/clickstream/aggregate-preferences/output</arg>
      </sqoop>
      <ok to="end" />
      <error to="kill" />
    </action>

    <kill name="kill">
      <message> Workflow failed. Error message
      [${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name="end" />
</workflow-app>
```

整体看来，该工作流如图 6-4 所示。

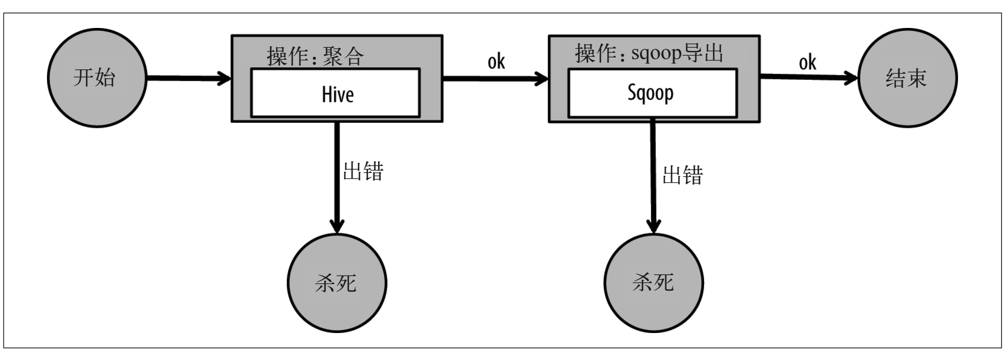


图 6-4：点对点式工作流

现在来深入分析该工作流中的每一个 XML 元素。

- `global` 元素包含所有操作都会用的全局配置，比如 JobTracker 与 NameNode 的 URI。`job-tracker` 元素在表意上稍有偏差。Oozie 使用相同的参数读取 JobTracker URI（使用 MR1 时）与 YARN ResourceManager URI（使用 MR2 时）。
- `start` 元素指工作流中第一个运行的操作。每个操作都有一个名称，而根据类型每个操作都有一组参数。
- 第一个 action 元素用来运行 Hive 操作。该元素获取了 Hive 配置文件与 HQL 脚本的位置，而后执行操作。记住，由于数据处理发布 Hive 命令的过程启动 Map 任务，能够在 Hadoop 集群的任意一个节点上运行，所以 Hive 配置文件与 HQL 脚本需要在 HDFS 中呈现，由此才能被运行 Hive 脚本的节点访问。在这个例子中，脚本 populate_agg_table.sql 通过 Hive 执行聚合操作，而且将聚合的结果存储到 HDFS 中，具体位置为 hdfs://nameservice1/etl/BI/clickstream/aggregate-preferences/output。
- `sqoop_export` 操作负责运行一个 Sqoop 输出任务，将聚合的结果从 HDFS 输出到 RDBMS。该过程需要通过一个参数列表进行，并且这个列表与从命令行调用 Sqoop 时使用的列表相对应。
- 另外，每个操作都包含接下来的方向，无论操作成功（到达工作流中下一个操作，或者结束），还是出现错误（到达 kill 节点，基于最后一个出现错误的操作产生恰当的错误消息，也能发送通知 JMS 消息或者邮件）。
- 注意，每个操作与工作流本身都有一个某版本的 XML 模式（如 `xmlns="uri:oozie:sqoop-action:0.4"`）。这规定了 XML 定义中可获得的元素。比如，`global` 元素只存在于 `uri:oozie:workflow:0.4` 及更高版本。如果使用一个旧版本的 XML 模式，那么你需要在每个操作中定义 `jobTracker` 与 `nameNode` 元素。对于目前可用的 Oozie 版本，我们建议使用可获得的最新模式。

6.8.2 扇出式工作流

当工作流中多个操作并行运行时，最常用到的是扇出式工作流模式，但是这里需要前面所有操作都完成再运行后一个操作。这也被称为分叉汇合（fork-and-join）模式。在本例中，我们想先运行一些初始的统计，然后再运行三个不同的数据查询以完成数据聚合。这三个数据查询——查询处方与替换药、去诊所看病以及化验结果——能够并行运行并且只依赖已经完成的初始统计结果。完成三个数据查询之后，我们想通过运行另一个 Hive 查询来得到总结报告。

以下为 Oozie 中的工作流定义。

```
<workflow-app name="build_reports" xmlns="uri:oozie:workflow:0.4">

   <global>
         <job-tracker>${jobTracker}</job-tracker>
         <name-node>${nameNode}</name-node>
         <job-xml>${hiveSiteXML}</job-xml>
   </global>

   <start to="preliminary_statistics" />
```

```
    <action name="preliminary_statistics">
       <hive xmlns="uri:oozie:hive-action:0.5">
          <script>${scripts}/stats.hql</script>
       </hive>
       <ok to="fork_aggregates" />
       <error to="kill" />
    </action>

    <fork name="fork_aggregates">
       <path start="prescriptions_and_refills" />
       <path start="office_visits" />
       <path start="lab_results" />
    </fork>

    <action name="prescriptions_and_refills">
       <hive xmlns="uri:oozie:hive-action:0.5">
          <script>${scripts}/refills.hql</script>
       </hive>
       <ok to="join_reports" />
       <error to="kill" />
    </action>

    <action name="office_visits">
       <hive xmlns="uri:oozie:hive-action:0.5">
          <script>${scripts}/visits.hql</script>
       </hive>
       <ok to="join_reports" />
       <error to="kill" />
    </action>

    <action name="lab_results">
       <hive xmlns="uri:oozie:hive-action:0.5">
          <script>${scripts}/labs.hql</script>
       </hive>
       <ok to="join_reports" />
       <error to="kill" />
    </action>

    <join name="join_reports" to="summary_report" />

    <action name="summary_report">
       <hive xmlns="uri:oozie:hive-action:0.5">
          <script>${scripts}/summary_report.hql</script>
       </hive>
       <ok to="end" />
       <error to="kill" />
    </action>

    <kill name="kill">
       <message> Workflow failed. Error message
                 [${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name="end" />
</workflow-app>
```

在该工作流中，第一个操作 `preliminary_statistics` 计算初始的统计结果。成功完成这

一步后，工作流推进至 fork 元素，随后三个操作（prescriptions_and_refills、office_visits 与 lab_results）由此开始并行运行。完成上述操作之后，控制（control）进入 join 元素。先前三个 Hive 操作全部完成之后，join 元素才会启动工作流。注意，这里假设合并之后的操作依赖属于 fork 操作的结果，所以只要有一个操作出现错误，整个工作流都会中断。三个 fork 操作成功完成，控制进入最后一个操作，即 summary_report。最后一个操作完成时，工作流将结束。

注意，因为所有的工作流都使用相同的 Hive 定义，所以 <job-xml> 只定义一次，具体位置为 <global> 部分。

图 6-5 为扇出式工作流。

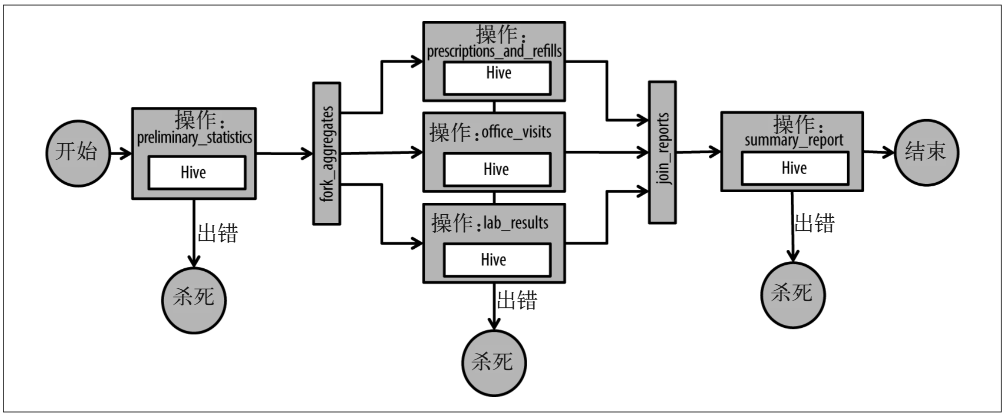


图 6-5：扇出式工作流

6.8.3　分支决策式工作流

需要基于前一个操作的结果选择下一个操作时，我们经常使用分支决策式工作流，比如主类 com.hadooparchitecturebook.DataValidationRunner（用于验证即将存入 Hadoop 的数据）中含有一些 Java 代码时。如果没有错误的话，根据这个验证，用户想通过运行主类 com.hadooparchitecturebook.ProcessDataRunner 来处理此数据。但是，如果出现错误，用户就要通过运行 com.hadooparchitecturebook.MoveOutputToErrorsAction，根据错误处理代码将错误信息存入 HDFS 上不同的目录中，并且获得目录报告。

以下为 Oozie 中实施这种工作流的 workflow.xml 文件：

```
<workflow-app name="validation" xmlns="uri:oozie:workflow:0.4">

   <global>
         <job-tracker>${jobTracker}</job-tracker>
         <name-node>${nameNode}</name-node>
   </global>

   <start to="validate" />
```

```
   <action name='validate'>
       <java>
            <main-class>com.hadooparchitecturebook.DataValidationRunner
            </main-class>
            <arg>-Dinput.base.dir=${wf:conf('input.base.dir')}</arg>
            <arg>-Dvalidation.output.dir=${wf:conf('input.base.dir')}/dataset
            </arg>
            <capture-output />
        </java>
        <ok to="check_for_validation_errors" />
        <error to="fail" />
    </action>

    <decision name='check_for_validation_errors'>
        <switch>
            <case to="validation_failure">
                ${(wf:actionData("validate")["errors"] == "true")}
            </case>
            <default to="process_data" />
        </switch>
    </decision>

    <action name='process_data'>
        <java>
            <main-class>com.hadooparchitecturebook.ProcessDataRunner</main-class>
            <arg>-Dinput.dir=${wf:conf('input.base.dir')}/dataset</arg>
        </java>
        <ok to="end" />
        <error to="fail" />
    </action>

    <action name="validation_failure">
        <java>
            <main-class>com.hadooparchitecturebook.MoveOutputToErrorsAction
            </main-class>
            <arg>${wf:conf('input.base.dir')}</arg>
            <arg>${wf:conf('errors.base.dir')}</arg>
            <capture-output />
        </java>
        <ok to="validation_fail" />
        <error to="fail" />
    </action>

    <kill name="validation_fail">
        <message>Input validation failed. Please see error text in:
                 ${wf:actionData("validation_failure")["errorDir"]}
        </message>
    </kill>

    <kill name="fail">
        <message>Java failed, error message[${wf:errorMessage
                 (wf:lastErrorNode())}]</message>
    </kill>
    <end name="end" />

</workflow-app>
```

图 6-6 为该工作流的一个总体概括。

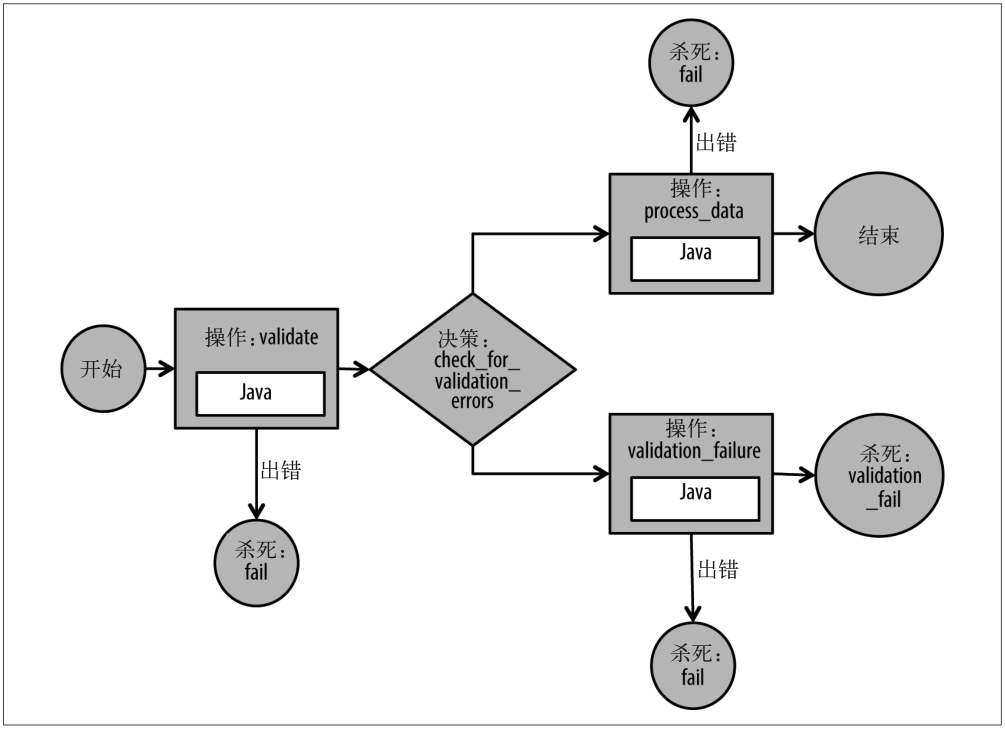


图 6-6：分支决策式工作流

该工作流中，第一个操作为 `validate`，运行了一个 Java 项目，用来验证一个输入数据集。根据 Oozie 配置中 `input.base.dir` 参数的值，其他一些参数被传送到 Java 项目中。注意，我们使用了 `<capture-output />` 元素来获取此操作的选择权，随后又会用在下一个决策点（即 `check_for_validation_errors`）上。Java、SSH 与 shell 操作都支持采集输出数据。在任何一种情况下，输出数据的格式都为 Java 属性的文件格式（http://bit.ly.property-file），每行包含一个键值对，行之间由换行符分隔。对于 shell 与 SSH 等才注意，输出数据应该写入标准的输出（stdout）。对于 Java 操作，应该收集 `Properties` 对象中的键与值，而且写入文件中。该文件名称必须从项目的 `oozie.action.output.properties` 系统变量中获取。

在前一个例子中，我们从操作的输出结果中检查了 `errors` 属性的值。下面这个例子展示的是如何为 Java 中采集的输出数据编写 `errors=false`。

```
File file = new File(System.getProperty("oozie.action.output.properties"));
Properties props = new Properties();
props.setProperty("errors", "false");

OutputStream os = new FileOutputStream(file);
props.store(os, "");
os.close();
```

然后，决策节点通过 `wf:actionData("_action_name_")["_key_"]` 引用采集到的输出数

据。其中 action 的名称为 _action_name_，而属性名称为 _key_。而后决策节点包含一个 switch 声明，如果在 validate 操作采集到的输出数据中 errors 属性设置为 true，或者设置为 process_data 数据操作，控制就会引向 validation_failure 节点。

process_data 操作只是调用了另一个 Java 项目来处理数据，同时将数据集的位置当作属性传送到项目中。出现验证错误时，validation_failure 操作调用另一个 Java 项目，并传送两个参数。但是这一次，参数作为命令行自变量（command-line argument）而不是属性传送（注意，此处缺少 -D <property=value> 语法）。用户期待本项目能够将验证错误相关的信息输入到目录中，并使用 capture-output 元素将该目录的名称返还到工作流。错误目录的名称作为 errorsDir 属性被传送出去，并被下一个操作读取，即 validation_fail，而后将目录的位置报告给用户。

6.9 工作流参数化

在不同的对象上运行相同的工作流也是一种时常出现的需求。比如，如果工作流使用 Sqoop 从关系型数据库输出一个表，而后使用 Hive 运行一些数据验证，那么用户可能想要在不同的表上使用同一个工作流。这种情况下，用户会想将表的名称设定为一个参数，并将该参数传送到工作流中。还有一种经常使用参数的情况发生在指定目录名称与数据时。

Oozie 能够让用户将参数指定为 Oozie 工作流中的变量或者协调器操作，而后，在调用工作流或者协调器时对这些参数设定值。除了能够在需要时迅速变更值而且不需要编写代码以外，用户还可以在不同的集群（常见的包括 dev、test 与 production）上运行相同的工作流，并且不用重新部署工作流。Oozie 工作流中指定的任一参数都能按照以下任何一种方式进行设置。

- 用户能够在 config-defaults.xml 文件中设定属性值，供 Oozie 随后获得不同配置属性的的默认值。本文件在 HDFS 中的位置必须在执行任务的 workflow.xml 文件后面。
- 用户能够指定 job.properties 文件中或者 Hadoop XML 配置文件中这些参数的值，并在启动工作流时使用 -config 命令行选择将其传送出去。但是，用户在编辑文件时可能会突然做出一些改变，而且不同用户使用的需求不同会导致同一文件会使用多次，所以这种方法也存在一些风险。
- 当使用 -D <property=value> 语法调用工作流或者协调器时，用户能够根据命令行指定这些参数的值。
- 从协调器中调用工作流时，用户能够传递这些参数。
- 用户能够在工作流定义中指定一列必需参数（也称作形式参数）。本列参数中可以包含默认参数。提交一个任务时，如果参数的值丢失，那么任务将撤销提交。用户能够在工作流的起始位置、之前指定该列参数。

```
<parameters>
    <property>
        <name>inputDir</name>
    </property>
    <property>
        <name>outputDir</name>
        <value>out-dir</value>
```

```
        </property>
    </parameters>
```

然后，这些参数作为 `${jobTracker}` 或 `${wf:conf('jobTracker')}` 被工作流访问。如果参数名称不是 Java 属性（也就是说，不仅仅包含 `[A-Za-z_][0-9A-Za-z_]*`），那么就只能由第二种方法访问，即 `${wf:conf('property.name')}`。

另外，我们已经看到在一个工作流中有多少操作将参数传递给调用的代码。shell 操作可能将参数传递给 shell 命令，Hive 操作可能会获取 Hive 脚本的名称，并将其作为参数或者用于参数化 Hive 查询的其他参数。而 Sqoop 操作可能会使用参数使 Sqoop 输入或输出数据参数化。以下代码将一个参数传递给一个 Hive 操作，然后在查询中使用。

```
<workflow-app name="cmd-param-demo" xmlns="uri:oozie:workflow:0.4">
   <global>
         <job-tracker>${jobTracker}</job-tracker>
         <name-node>${nameNode}</name-node>
   </global>

    <start to="hive-demo" />

    <action name="hive-demo">
        <hive xmlns="uri:oozie:hive-action:0.5">
            <job-xml>${hiveSiteXML}</job-xml>
            <script>${dbScripts}/hive1.hql</script>
            <param>MYDATE=${MYDATE}</param>
        </hive>
        <ok to="end" />
        <error to="kill" />
    </action>

    <kill name="kill">
        <message>Action failed, error message
        [${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>

    <end name="end" />
</workflow-app>
```

在这个工作流中，参数 `${MYDATE}` 随后作为参数 `MYDATE` 传递到 Hive 查询。然后该 Hive 查询使用这个参数，如例 6-3 所示。

例 6-3　hive1.hql

```
INSERT INTO test SELECT * FROM test2 WHERE dt=${MYDATE}
```

6.10　Classpath定义

Oozie 操作通常为 Oozie 执行的小型 Java 应用。也就是说，执行操作时，你必须在 Classpath 中获取操作的依赖关系。Oozie 能够让用户使用预定义操作（该位置被称为 sharelib）为库定义一个共享的位置，所以能够处理一些依赖关系管理的问题。而且，Oozie 也为开发者提供了多种方法，根据不同的应用将不同的库增加到 Classpath。这种库

能够包含用于 Java 操作的依赖关系，或者用于 Hive 操作的用户自定义函数。

> 以下章节改编自 Cloudera 博客中的帖子：How-to: Use the ShareLib in Apache Oozie（http://blog.cloudera.com/blog/2014/05/how-to-use-the-sharelib-in-apache-oozie-cdh-5/）。

所有的 Hadoop JAR 都会自动包含在 Classpath 中，而且 sharelib 目录包含用于内置操作的 JAR（如 Hive、Pig、Sqoop，等等）。

sharelib 目录通过运行 `oozie-setup sharelib create -fs HDFS:\\host:port` 创建。如果想让工作流使用 shareLib，在 job.properties 文件指定 `oozie.use.system.libpath=true` 即可。在任务中，Oozie 通过一些必要的操作将 JAR 包含在 shareLib 中。

如果需要为一个操作增加自定义的 JAR（包括用于 MapReduce 或 Java 操作的 JAR、JDBC 驱动器，或者用于 Sqoop 操作的连接器），可以采用以下几种方法。

- 在 job.properties 中设定 `oozie.libpath=/path/to/jars,another/path/to/jars`。如果有很多工作流都需要同一个 JAR，那么这种方法会很有用。用户可以将其放置在 HDFS 中，而且在多个工作流中使用。工作流中所有的操作都能获得该 JAR 包。
- 创建一个名称为 lib 的目录，将其放置在 HDFS 中 workflow.xml 文件的附近，并且把 JAR 放置在该位置。如果你有一些 JAR，但是只有一个工作流，这种方法会很实用。
- 在一个操作中指定 `<archive>` 标签，并且路径指向一个单一的 JAR。用户能够指定多个 `<archive>` 标签。如果 JAR 包只用于特定的操作，而不是工作流中所有的操作，那么这种方法会很有用。
- 将 JAR 增加到 sharelib 目录。但是这种方法一般不予推荐，原因有两个：首先，JAR 会被包含到每一个工作流中，因此会导致意外的结果；第二，如果更新 sharelib，JAR 将会被移除并且需要用户记住再次添加。

6.11 调度模式

我们已经知道，不同的操作，以及操作相关的决策都能放在一个工作流中，然后由一个自动化流程系统运行。但是，工作流在事件上是不可知的（time-agnostic），也就是说，工作流本身没有何时或如何运行的概念。用户可以手动运行工作流，但是如果可以确定预测结果是真的，工作流也能进行调度并运行。Oozie 利用了协调器，使用户通过一个 XML 文件（或者通过 Hue Oozie UI）来调度工作流，而在通过 Azkaban UI 执行工作流时，Azkaban 提供了调度选择。

Oozie 的情况是，协调器在名称为 coordinator.xml 的文件中定义为 XML 格式，然后传递到 Oozie 服务器中用于提交任务。Oozie 协调器引擎在 UTC 中工作，但是对于协调器应用支持不同的时区以及夏令时（Daylight Saving Time，DST）。

尽管 Oozie 支持相当多的调度使用案例，但目前最新版本的 Azkaban 只支持第一个被称为依频次调度（frequency scheduling）的使用案例，我们随后将对其讨论。

6.11.1 依频次调度

依频次调度是最简单的调度，能够使用户周期性地执行一个工作流。比如，如果用户想每 60 分钟执行一个工作流，进行每小时的数据聚合，那么可以使用如下 Oozie coordinator.xml 文件。

```
<coordinator-app name="hourly-aggregation" frequency="${coord:minutes(60)"
start="2014-01-19T08:00Z" end="2014-01-20T08:00Z" timezone="America/Los_Angeles"
xmlns="uri:oozie:coordinator:0.4">
   <action>
      <workflow>
         <app-path>hdfs://nameservice1/app/workflows/hourly-aggregation
         </app-path>
      </workflow>
   </action>
</coordinator-app>
```

在该例以及随后一个例子中，结尾处的后缀 Z 代表时间戳在 UTC 中。

前面的协调器称为 hourly-aggregation，每 60 分钟运行一次 hourly-aggregation 工作流（frequency 属性的默认单元为分钟）。该工作流第一次的运行时间是 2014 年 1 月 19 日上午 8 点（UTC 时间），最后一次运行时间为 2014 年 1 月 20 日上午 7 点（UTC 时间），这是结束前一个小时，结束时未运行。用户期望运行的工作流定义位于 HDFS 目录 hdfs://nameservice1/app/workflows/hourly-aggregation 下、名称为 workflow.xml 的文件中。由于 start 与 endfield 正好相差一天（即 24 小时），所以这个工作流运行了 24 次。Oozie 也支持灵活性与可读性都非常好的 cron。

6.11.2 时间或数据触发式

通常，工作流需要在特定时间运行，但是这样的话只能获得数据集中一个特定的数据集或者一个特定的分区。如果存在一个特定数据集或者分区，那么 Oozie 协调器能够只触发一个工作流。如果不存在这个数据集，协调器则会定期检查该数据集是否存在，直到超出特定的期限。如果指定 timeout=0，而且数据不存在，那么协调器会舍弃该工作流。并且，协调器不会再次检查数据是否存在，直到整个协调器被按照 coordinator-app 中所列的 frequency 再次触发。默认的超时时间为 -1，这时 Oozie 服务器会使协调器处于队列等候状态，直所有的要求得到满足——换句话说，直到能够获取要求的数据集。

我们假设 Hadoop 集群与 Oozie 服务器都在加利福尼亚。比如你想每天从凌晨 1 点开始，按小时收集前一天的数据。如果上游问题使数据集延迟，你只想在数据集到达时触发工作流。在 Oozie 中实现这种协调器的例子如下所示。

```
<coordinator-app name="hourly-aggregation" frequency="${coord:days(1)}"
start="2014-01-19T09:00Z" end="2015-01-19T10:00Z" timezone="America/Los_Angeles"
xmlns="uri:oozie:coordinator:0.1">
   <dataset name="logs" frequency="${coord:days(1)}"
           initial-instance="2014-01-15T06:15Z" timezone="America/Los_Angeles">
    <uri-template>
      hdfs://nameservice1/app/logs/${YEAR}${MONTH}/${DAY}
    </uri-template>
```

```
      <done-flag>_DONE</done-flag>
    </dataset>
      <input-events>
      <data-in name="input" dataset="logs">
        <instance>${coord:current(-1)}</instance>
      </data-in>
    </input-events>
     <action>
        <workflow>
           <app-path>hdfs://nameservice1/app/workflows/hourly-aggregation
           </app-path>
        </workflow>
     </action>
  </coordinator-app>
```

从太平洋时间的 2014 年 1 月 19 日上午 1 点开始（在 UTC 中为 09:00），本协调器每天都会触发。datasetfield 描述了在工作流运行之前需要检查的数据集。

> 尽管协调器在太平洋时区运行，但我们在 UTC 中也标记了开始与结束时间。Oozie 在 UTC 中执行每个操作，而且在其他时区指定开始与结束时间会导致提交错误。计算频率时，时区域只用于 DST 值。

frequency 标记了频率，单位为分钟，此时数据集被周期性写入。Oozie 使用这种方法检查数据集过往运行第 *n* 次的情况。我们使用 ${coord:days(1)}，而没有将 frequency 指定为 1440（一天的分钟数）。原因在于当 DST 开始或结束时，一天的分钟数会发生变化。initial-instancefield 指数据集第一次出现。在 initial-instance 之前出现的数据集会被忽略。HDFS 上指向数据集目录的路径由 uri-templatefield 定义。URI 将分解为 /app/logs/2014/06/19、/app/logs/2014/06/20，等等。尤其是，协调器将查找 HDFS 目录中是否存在名称为 _DONE 的文件，以确定目录是否准备好处理数据。本文件的名称由 done-flagfield 提供。如果将 done-flag 设定为一个空字符串，Oozie 将自己查找是否存在目录。URI 模板中使用的 YEAR、MONTH 与 DAY 变量基于 UTC 中的实际时间，如果有需要，Oozie 会自动执行会话。如果协调器应用首先在 2014-01-19T09:00Z（即，UTC 时间 2014 年 1 月 19 日上午 9 点）触发，那么 YEAR、MONTH 与 DAY 分别映射为 2014、01 与 19。注意，这也是协调器应用的触发时间（加利福尼亚上午 9 点），但不一定是工作流运行的时间（可能需要等到可以获得输入数据）。

> 因为 uri-template 只依赖 YEAR、MONTH 与 DAY，所以 initial-instance（如 06:15）的小时与分钟部分并不重要。无论是 06:15 还是 06:16，都会映射到同一个时间，而且第一次运行的 URI 相同。只要最初的运行时间映射到 UTC 时间中的相同的 URI（YEAR、MONTH、DATE 与 UTC 时间中评估的相似变量），initial-instance 的小时与分钟部分在本案例中并不重要。

input-events field 定义执行工作流所需要的输入数据。在这个例子中，我们将日志数据设定为需要的输入数据。我们也指定这个日志数据集所需要的实例。此时，我们想获取一天前的实例，所以指定 ${coord:current(-1)}。这个标准化的实例会被转化为 UTC，而后用

来自动化地填充变量 YEAR、MONTH、DAY，等等。

由于先前的例子没有指定 timeout，所以如果无法获取数据集实例，Oozie 服务器将无限期等待，直到可以获得数据集实例运行工作流。这种情况下，为协调器任务设置并发、节流与执行策略非常很重要。并发（concurrency）指对于同一个任务来说，有多少个实例能在任一给定时间运行，其默认值为 1。节流指同一时间等待执行的任务数量。执行指任务中两个或多个实例在排队等待时下达的执行命令，默认 FIFO（First In First Out，先进先出，即最先触发的任务最先执行）。其他选择包括 LIFO（Last In First Out，后进先出，即最后触发的任务最先进行）与 LAST_ONLY（只运行最后触发的任务）。

如果工作流仅指某天某个时间的数据，那么多个任务在 FIFO 命令中同时运行比较适合。包含这类设置的 coordinator.xml 文件如下所示。

```
<coordinator-app ...>
  <controls>
    <timeout>1440</timeout>
    <execution>FIFO</execution>
    <concurrency>5</concurrency>
    <throttle>5</throttle>
  </controls>
  <dataset ....>
.
.
</coordinator-app>
```

另外，如果用户只想让 Oozie 服务器检查是否可以获得数据集，并且在放弃之前最多维持 2 个小时，那么可以将 timeout 设置为 120 分钟。

包含这类设置的 coordinator.xml 文件如下所示。

```
<coordinator-app ...>
  <controls>
    <timeout>120</timeout>
  </controls>
  <dataset ....>
.
.
</coordinator-app>
```

如果用户根本不想等到可以获得数据集实例，也就是说数据已然无法获得，那么你可以将 timeout 设置为 0。由此，在无法获得数据的情况下，协调器就能直接退出而不会一直等到超时。

同样，你也可能通过指定一个以上的数据依赖关系依赖多个数据集。目前为止，我们一直在讨论每日写入的数据集。现在让我们来看一个每小时写入数据集的例子。如果按照每小时写入数据集，而且需要处理 24 小时的数据才能完成最后一个小时的任务，那么用户可以加强开始与结束时所有数据集实例的出现。假设这个协调器在每小时的第 10 分钟触发，那么该协调器任务如下所示。

```
<coordinator-app name="hourly-aggregation" frequency="${coord:hours(1)}"
start="2014-01-19T09:10Z" end="2015-01-19T10:10Z" timezone="America/Los_Angeles"
```

```
xmlns="uri:oozie:coordinator:0.1">
.
.
.
<input-events>
     <data-in name="coordInput1" dataset="input1">
         <start-instance>${coord:current((coord:tzOffset()/60) - 24)}
         </start-instance>
         <end-instance>${coord:current(coord:tzOffset()/60) - 1}</end-instance>
     </data-in>
  </input-events>
.
.
.
```

在这里，我们使用 ${coord:current((coord:tzOffset()/60) - 24)} 作为开始实例。我们暂时不用看 -24 部分。按照分钟来算，coord:tzOffset() 导致名义上的时区与 UTC 之间出现偏差。按照小时来算，coord:tzOffset()/60 同样也带来了偏差。对于加利福尼亚来说，这里通常在正常时间内为 –8 而在 DST 之内为 –7。如果我们只使用 ${coord:current()}，那么与触发时间 2014-01-19T09:10Z 相对应的实例为 YEAR=2014, MONTH=01, DAY=19, HOUR=09。但是，我们的数据集是在太平洋时区内产生的，既然加利福尼亚的时间才到上午 1 点，那么实例并不存在。为了获取相关实例，我们不得不考虑时区偏差，并执行 ${coord:current(coord:tzOffset()/60)}。因为想要获取包含最后一个小时的 24 小时实例，我们减去 1，得到最后的实例，而减去 24，则可以得到最初的实例。要注意的是，先前的例子并不关心时区偏差，原因在于我们的数据集是每日写入的。从技术上来说，我们在那里可以使用 ${coord:current(coord:tzOffset()/1440) - 1}（1440，按照每日写入数据集），而不是只使用 ${coord:current(-1)}。但我们从地理课上学过，所有的时区偏差通常都小于 1440 分钟（24 小时），所以那个值会一直为 0。

用户可能需要将小时时间戳传递到工作流，而后用于工作流。Oozie 提供了一系列用于协调器的函数，能够产生并格式化时间戳，将其传递给工作流。此类协调器如下所示。

```
<coordinator-app name="hourly-aggregation" frequency="${coord:hours(1)}"
start="2014-01-19T09:10Z" end="2015-01-19T10:10Z" timezone="America/Los_Angeles"
xmlns="uri:oozie:coordinator:0.1">
   <action>
      <workflow>
         <app-path>hdfs://nameservice1/app/workflows/hourly-aggregation
         </app-path>
         <configuration>
          <property>
                <name>dateHour</name>
                <value>${coord:formatTime(coord:nominalTime(),'yyMMdd_HH')}
          </property>
         </configuration>
      </workflow>
   </action>
</coordinator-app>
```

当协调器在 UTC 中被触发时，coord:nominalTime() 返回对应的时间。这种情况下，工作

流会接受一个名称为 `dateHour` 的参数，在目前的小时之内该参数的值是一个格式化的字符串。然后这个参数会用于聚合操作，一如 6.9 节中所示的参数。

6.12 执行工作流

如果想要执行一个工作流或者协调器，那么你需要首先把 XML 定义放置到 HDFS 中。同样也要放置工作流在 HDFS 中所依赖的所有 JAR 包或文件，如 6.10 节中所述。接下来要定义一个属性文件，通常命名为 job.properties。这个文件应该包含所有你想在工作流中使用的参数。你必须指定 `oozie.wf.application.path`，即工作流定义 XML 文件的 URL。通常为 `${nameNode}/user/${user.name}/myworkflow`，往往会指定以下的内容。

- `nameNode`

  NameNode 的 URL，应该与 `hdfs://hadoop1:8020` 相似。

- `jobTracker`

  JobTracker 的 URL，与 `hadoop1:8021` 相似。

另外，如果工作流包含 Hive、Pig 或 Sqoop 操作，那么 Oozie 会要求访问这些库。这些可能已经作为 Oozie 安装（指 Oozie sharelib）的一部分包含在了 HDFS 中，但是用户需要指定想要使用的部分，将 `oozie.use.system.libpath=true` 增加到自己的属性文件中，详见 6.10 节。

执行协调器也需要包含 `oozie.coord.application.path`，即协调器定义 XML 的 URL（比如，`${nameNode}/user/${user.name}/myworkflow/coord.xml`）。

6.13 小结

如果你在 Hadoop 上开发应用，用于管理多种操作间任何一种不常见的依赖关系，那么你需要一个工作流管理系统。如果你已经拥有一个企业自动化流程引擎，那么我们建议你开发该引擎，使其能够支持 Hadoop。你可以随意选择，但是我们不建议对任何不常见的关系这么做。在后一种情况下，你应该使用一种现存的引擎，如 Oozie，这一点我们已经详细讨论过。具体如何选择，需要参考实际的使用情况。

如果你的大多数自动化流程位于 Hadoop 中间，而且赞同将工作流与协调器定义放置于 HDFS 中，那么 Oozie 会非常实用。如果你使用的是包含 Oozie 的发行版，则更是如此。它还提供安装与管理支持。其他可以开发的开源选择包括 Azkaban、Luigi 与 Chronos。

第 7 章

Hadoop近实时处理

一直以来，Hadoop 在整个开发演进的过程中都扮演着批处理系统的角色。最常见的处理范式就是将数据加载至 Hadoop，接下来通过批处理任务（通常采用 MapReduce 实现）处理数据。这类处理的常见场景是抽取–转换–加载（Extract-Transform-Load，ETL）数据处理流水线——将数据加载到 Hadoop 中，按照某种方法对数据进行转化操作，以符合业务的需求，而后将数据加载到用于进一步分析的系统中（比如关系型数据库）。

Hadoop 高效并行处理海量数据的功能为用户带来了很多益处。但是有些情况对于数据处理有着更高的实时要求，更准确地说，应该是近实时要求，我们稍后会简单介绍。要在数据到达时对其处理，而不是批处理数据。这种类型的数据处理通常称作流处理（stream processing）。

幸运的是，随着新工具集成到 Hadoop 生态系统中来，这种近实时数据处理的需求也有了满足的办法。这些流数据处理工具包括 Apache Storm、Apache Spark Streaming、Apache Samza 等系统，甚至 Apache Flume 通过 Flume 拦截器（Interceptor）也能够提供支持。与 MapReduce 之类的批数据处理系统不同，这些工具能够让用户创建各种数据处理流程，从而处理持续到来的数据。这些流程只要在运行，就会一直进行数据处理，而批数据处理只能在固定的数据集上操作，两者正好相反。这种类型的系统中，在 Hadoop 上应用最广泛的是 Storm 与 Spark Streaming，本章将讨论这两种系统。

下面，对于以上类型的数据处理，举几个例子。

- 社交媒体消息分析，例如：在某个社交媒体网站上动态探测某用户的更新趋势。
- 财经消息分析，例如：检测某用户账户的异常活动。
- 视频游戏使用消息分析，例如：观察用户行为，发现某些玩家的作弊行为并采取防范措施。
- 机器数据消息分析，例如：应用日志中出现异常或错误时发出警报。

Apache Kafka 之类的工具怎么样

Kafka 这样的工具在近实时数据处理中的作用可能令人困惑。实际上，Kafka 是一种分布式消息总线。它的主要用途是在架构上提供可靠的消息传递，以保证事件的快速消费需求。Kafka 是一种强大的工具，但是并不能在数据传输过程中提供数据转换、报警或者计数的功能。

重要的一点是，尽管 Kafka 不是一种流处理系统，但在很多架构中，Kafka 经常是涉及流处理时的关键部分。正如我们在讨论 Storm 时所谈到的，Kafka 提供的数据传输保证让 Kafka 本身成为了与 Storm 互补的良好选择。在将事件采集到 Hadoop 中时，Kafka 也常作为整个架构的一部分。

另一点需要注意的是，尽管大家常认为 Flume 是一种采集机制，但我们也可以利用 Flume 拦截器的功能进行事件级别的操作。Flume 拦截器能够让我们执行一些很常见的流处理活动，如数据补充与验证或提醒。7.5 节将进一步探讨这些内容。

开始之前，我们应该先注意几个词。第一个词是实时。这个词现在被滥用了，而且通常定义都很不准确。为了更好地进行讨论，我们在本章提到的数据处理都采用更为准确的定义，即近实时——几秒到几百毫秒之内进行的数据处理。如果你的应用要求数据处理快于 5~100 毫秒，那么本章讨论的工具不符合你的需求。

另外，本章不会讨论以下内容：Cloudera Impala、Apache Drill 或 Presto 这样的查询引擎。NRT 上下文经常谈及这些系统，但是它们实际上进行的是低延迟的大规模并行数据处理（Massively Parallel Processing，MPP）。在探究 Hadoop 上的数据处理时，第 3 章曾讨论过这些查询引擎。

图 7-1 展示了这些工具在一个实时上下文中应该是怎样的。浅灰色的格子表示本章将要讨论的系统。深灰色的格子表示这些工具按照执行时间排列的先后位置，只是本章并没有专门提到。

请注意左边标记着自定义的格子。尽管有时 Storm 或 Flume 数据处理的速度接近或快于 50 毫秒，但通常来说达到这个级别的处理需要在 C/C++ 中实现自定义应用。

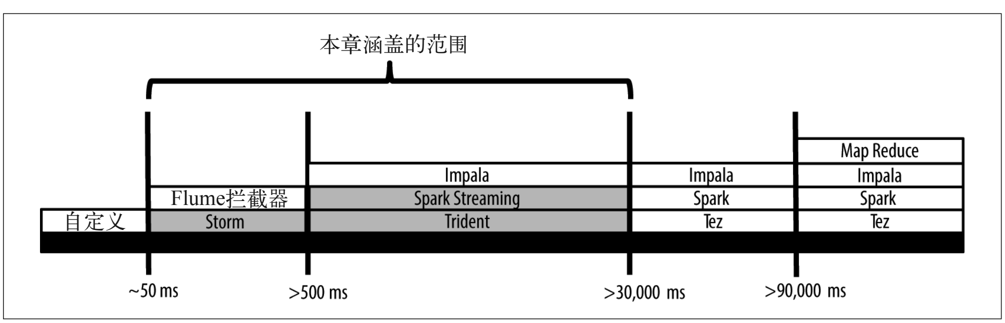


图 7-1：近实时处理工具

因为 HBase 通常作为流处理的持久层使用，而且在决策支持或者数据扩充方面能够提供一些功能性的帮助，所以我们也会探讨如何让 HBase 适用于 NRT 架构。

我们首先了解一下流处理引擎 Storm（以及相关的 Trident 项目）与 Spark Streaming 的概况，随后比较一下这些工具。然后，我们将讨论用户什么时候可能会想要使用流处理工具，以及如何选择工具。

7.1 流处理

我们之前就注意到了，流处理指一种系统，这种系统能够不断地持续处理到来的数据，直到应用停止。流处理不是一个崭新的概念，早在 Hadoop 出现之前，执行流处理的应用就已经存在了，包括商业项目，如 StreamBase，以及类似于 Esper 的开源工具。更新型的工具（如 Storm、Spark Streaming 与 Flume 拦截器）能够与 Hadoop 集成到一起，执行分布式的流处理。

针对这些数据流框架，我们将探讨它们如何处理以下常见的功能。很多 NRT 使用场景都需要这些功能。

- *聚合*

  即计数器管理功能，经典的单词计数就是一个简单的例子。计数器在很多情形下都会用到，常见应用包括报警和欺诈检测。

- *窗口平均*

  在给定数量的事件或者给定的时间窗口，计算一个平均数。

- *记录级别的数据扩充*

  能够根据规则或者外部系统（如 HBase）的内容，修改一条给定的记录。

- *记录级别的报警或验证*

  能够基于系统中单个事件的到达情况，进行报警或抛出警告。

- *临时数据的持久化*

  在数据处理期间能够存储状态——如，通过计数或平均数持续计算结果。

- *支持 Lambda 架构*

  Lambda 架构这个概念也遭到了滥用。我们在下文中提供了更完整的定义。简单来说，我们认为它填补了不精确的数据流结果与更精确的批数据计算之间的鸿沟。

- *高级别的函数功能*

  支持多种函数，其功能包括分类、分组、分区与合并数据集以及数据集的子集。

- *与 HDFS 集成*

  与 HDFS 集成的简易程度与成熟度。

- *与 HBase 集成*

  与 HBase 集成的简易程度与成熟度。

Lambda 架构

本章提到了 Lambda 架构，下面我们简单看一下它究竟是什么。Nathan Marz 与 James Warren 定义了 Lambda 架构，在他们的书 Big Data 中有详细的描述。这是一种常用的框架，用于可伸缩与容错的海量数据处理。

Lambda 架构通过将架构分片成三层，为实时计算任意数据的任意函数提供了可能。

- Batch 层

  这一层存储了数据集的主副本。这个主数据是一种不可变的副本，复制了进入系统的原始数据。批量层也对 Batch 视图作出预计算，这也是对数据查询的基本预计算。

- Serving 层

  该层索引、加载 Batch 视图，并将其用于低延迟的查询。

- Speed 层

  这是架构中基本的近实时层。在数据到达系统时，这层创建了数据视图。这是与本章联系最密切的层，原因在于这层也许能够通过流数据处理系统（如 Storm 或 Spark Streaming）实现。

新数据会运送到 Batch 层与 Speed 层。在 Batch 层中，新数据会添加到主数据集中；而在 Speed 层中，新数据会用于近实时视图的增量更新。在查询时，来自两个层的数据会被合并。当 Batch 层和 Serving 层的数据均可用时，Speed 层的数据就可以丢弃了。

这种设计提供了一种可靠的、容错的方法，能用于要求低延迟数据处理的应用。Batch 层中数据的权威版本意味着，即使在相对不那么可靠的 Speed 层中出现错误，系统状态也能重建。

我们首先大致了解一下 Storm，并检查它对于流处理系统标准的符合情况。讨论过 Storm 之后，我们将介绍 Trident。Trident 是一种 Storm 之上的抽象，提供了高一层的功能，也通过小的批处理化加强了 Storm 的核心架构。

7.2 Apache Storm

Apache Storm 是一种开源系统，用于流数据的分布式处理。Hadoop 用户会觉得 Storm 的很多设计原则都很熟悉。Storm 架构的创建有以下几个目的。

- 开发与部署的简易性

  Hadoop 上的 MapReduce 在实现分布式批数据处理时减少了复杂性，与此相同，Storm 减少了创建数据流应用所需要的任务。Storm 使用简单的 API 与数量有限的抽象来实现工作流，易于配置和部署。

- 可伸缩性

  与 Hadoop 相同，Storm 能够有效地并行处理海量消息，而且能够通过添加新节点简单地进行扩展。

- 容错

  与 Hadoop 相同，Storm 在架构设计上允许错误的发生。与 Hadoop 任务中的任务相同，如果处理或者节点出现错误，Storm 数据处理是可以多次启动的，而且任务失败时将自动重启。但是需要注意，本地的持久值（如计数与滚动平均值）不支持容错。这类容错需要一个外部的持久化系统支持。我们随后将看一下 Trident 是如何解决这个问题的。

- 数据处理保证

  Storm 保证每个传过系统的消息都会得到处理。出现错误时，消息会重新发送，保证处理。

- 广泛的项目语言支持

  实现工作流时，Storm 使用 Apache Thrift 提供了语言的可移植性。

关于 Storm 的完整概述不在本书的范围之内，所以我们会把重点放在 Storm 与 Hadoop 的集成上面。如欲了解更多，请参阅 Storm 文档（http://storm.apache.org/documentation/Home.html）。想在实用层面了解 Storm，请参阅 Sean T.Allen 等编著的 *Storm Applied* 。

Microbatch 与 Streaming 的对比

我们暂且停下，先介绍一下本章使用次数比较多的一个词：Microbatch。Storm 是纯数据流工具，而 Spark Streaming 与 Trident 提供了一个 Microbatch 模型，能够简单地将事件分组，使其成为离散事务性的批量数据，然后处理。在讨论不同工具与使用案例时我们将详细讨论这个概念。重要的是，这里需要注意，与 Streaming 相比，Microbatch 未必一直是一种更好的方法。设计架构时需要做出很多决策，与之类似，选择 Microbatch 还是 Streaming 依赖于使用案例的具体情况。在降低延迟的情况下，Streaming 是合适的选择。另外一些情况下，如果想要更好地支持只进行一次的数据处理，那么更应该选择 Microbatch。

7.2.1 Storm高级架构

图 7-2 为 Storm 架构中的组件。

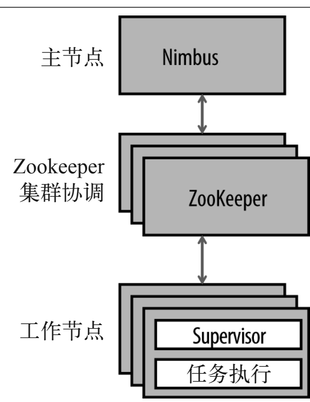


图 7-2：Storm 高级架构

如图 7-2 所示，在一个 Storm 集群中存在两种类型的节点。

- 主节点运行一个名称为 Nimbus 的数据处理。Nimbus 负责在集群周围分配节点，将任务分配给机器并监控是否出现错误。你可以认为 Nimbus 数据处理与 MapReduce 框架中的 JobTracker 相似。
- Worker 节点运行 Supervisor 后台程序。Supervisor 监听并服从分配给该节点的任务，也负责启动或停止 Nimbus 分配给节点的数据处理进程。这与 MapReduce 中的 TaskTrackers 相似。
- 另外，ZooKeeper 节点能够协调 Storm 数据处理任务，也能存储集群状态。在 ZooKeeper 中存储集群状态能够使 Storm 数据处理自身无状态，支持 Storm 数据处理出现错误或者重启，而不会影响集群的工作。注意，在这种情况下，“状态”指集群的操作状态，而不是数据在集群中的传输状态。

7.2.2 Storm拓扑

处理流数据的应用由 Storm 拓扑实现。拓扑是一种计算图，拓扑中的每个节点都包含数据处理逻辑。节点之间的联系定义了数据在节点之间传递的方式。

图 7-3 为一个简单的 Storm 拓扑例子，即经典的词汇计数。如表中所示，spout 与 bolt 是 Storm 的操作原语，提供了处理流数据的逻辑，而 spout 与 bolt 网络组成了 Storm 拓扑。spout 提供了数据流的来源，从类似于消息列表和社交媒体消息之类的来源获取输入数据。bolt 消耗一个或多个数据流（无论是来自于 spout 还是 bolt），执行一些数据处理，并选择性地创建新的数据流。我们随后会进一步讨论 spout 与 bolt。

注意，每个节点都是并行运行，而且一个拓扑的并行化程度是可配置的。一个 Storm 拓扑将一直运行下去，除非用户将其撤销，而且 Storm 会自动化重启失败的处理任务。

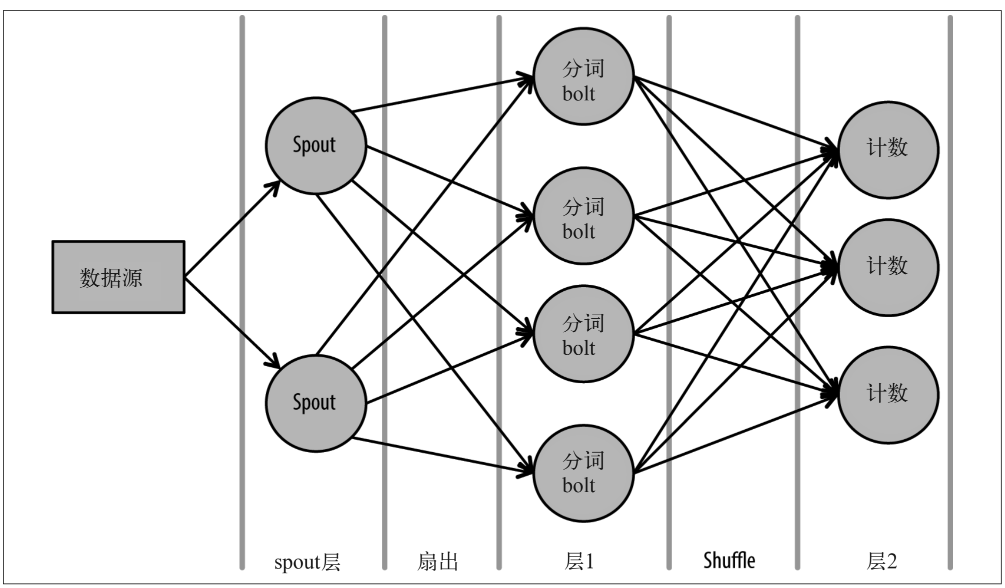


图 7-3：Storm 单词计数拓扑

我们将在后面的例子中详细讲解 Storm 的代码。以下就是图 7-3 中启动单词计数拓扑用到的代码。

```
builder.setSpout("spout", new RandomSentenceSpout(), 2);
builder.setBolt("split", new SplitSentence(), 4).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 3).
    fieldsGrouping("split", new Fields("word"));
```

我们很快就将看到，启动 Storm 拓扑的程序与本章讨论的其他系统 [如 Trident（详见 7.3 节）与 Spark Streaming（详见 7.4 节）] 大不一样。在 Storm 中，创建一个拓扑是为了解决一个问题。在 Trident 与 Spark Streaming 中，你要表达的是如何解决一个问题，而拓扑在场景之后创建。这种方法与 Hive 在 MapReduce 上提供一个抽象层相似。

7.2.3　元组及数据流

元组及数据流为通过 Storm 拓扑流动的数据提供了抽象。元组在 Storm 中提供了数据模型。一个元组是一系列有序、有名称的值（见图 7-4）。元组中的 field 可以是任何一种类型的。Storm 提供了初始类型、字符串与字节数组。用户可以通过一个自定义的 Serializer 使用其他类型 。一个拓扑中的每个节点都为自己产出的元组声明 field。

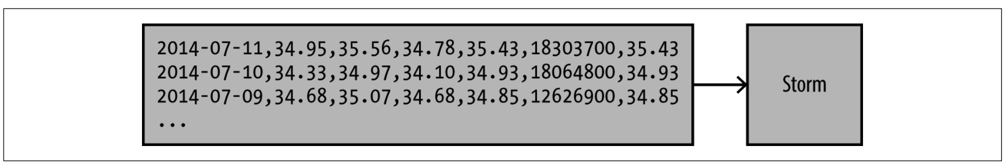


图 7-4：Storm 元组

数据流是 Storm 中的核心抽象。在一个拓扑中，一个数据流是任何两个节点之间的无约束元组序列（见图 7-5）。一个拓扑可以包含任意数量的数据流。

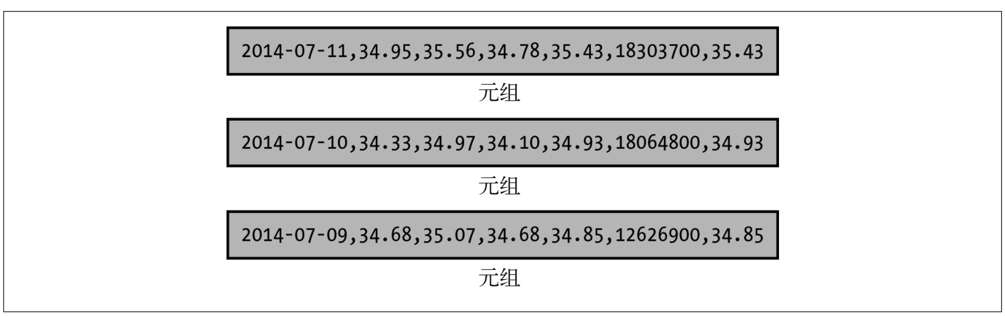


图 7-5：Storm 数据流

7.2.4　spout和bolt

正如我们先前提到的，Storm 架构中另外的核心组件是 spout 与 bolt，它们在 Storm 拓扑中提供数据处理。

- spout

  如前所述，spout 在一个拓扑中提供了数据流的来源。如果你对 Flume 比较熟悉，那么你可以将 spout 当作松散的 Flume 数据源。spout 从一些外部来源读取数据，如社交媒体信息或者消息队列，并将一个或多个数据流发送到一个拓扑中处理。

- bolt

  在一个拓扑中，bolt 消耗来自 spout 或上游 bolt 的数据流，在数据流中的元组上进行数据处理，并发出零至多个元组到下游 bolt 或外部系统，如一个数据持久层。与串联 MapReduce 任务相似，复杂的数据处理通常会由一个 bolt 链实现。

值得注意的是，bolt 在每个时间点处理一个单一的元组。回想一下我们在对比 Microbatch 与 Streaming 时讨论的内容，在不同的使用情况下，这种设计可能有优势，也可能有劣势。我们随后会讨论这一内容。

7.2.5 数据流分组

在一个拓扑中，数据流分组确定了 Storm 在任务集之间发送元组的方法。Storm 提供了一些分组方法，你也可以实现自定义的数据流分组。常见的数据流分组方法如下所述。

- Shuffle 分组

  元组被随机发送到一个 bolt 实例，同时保证每个 bolt 实例都将接收到同样数量的元组。能够独立处理每一个元组时，适合选择 Shuffle 分组。

- field 分组

  根据元组中的一个或多个 field 控制元组被发送到 bolt 的方式。对于给定的一组值，元组总会发送到同一个 bolt。与 MapReduce 数据处理相比，这种方法在某些方面与分区相似，分区的目的是采用相同的键将值发送到相同的 `Reducer`。需要将元组看作一个组时（比如，聚合值时），可以使用 field 分组。值得注意的是，对于 MapReduce 来说不同的键只能发送到不同的 `Reducer`。但是与 MapReduce 不同，使用 Storm 时，一个单一的 bolt 完全可以接受与其他组相关的元组。

- 全体分组

  对于所有参加的 bolt，复制数据流的值。

- 全局分组

  将一个完整的数据流发送到一个单一 bolt，与 MapReduce 任务类似（将所有的值发送到一个单一的 `Reducer`）。

想看到数据流分组的完整介绍，请参阅 Storm 文档（http://storm.apache.org/documentation/Concepts.html）。

数据流分组是 Storm 的一个重要特点，而且与定义拓扑的能力组合到了一起。正因如此，与 Flume 系统相比，Storm 更适合计数与滑动平均等任务。Flume 能够接收事件并执行事件增强与验证，但是不包含一个良好的系统，无法对那些事件分组，在不同的分区分组中处理。注意，Flume 具有分区的功能，但是需要在物理层次上定义，同样的情况 Storm 要在逻辑拓扑层次上进行定义，这样更为易于管理。

7.2.6 Storm应用的可靠性

Storm 在处理元组时提供了不同程度的保证。虽然程度不同，但实现的复杂度并没有增加太多。以下为不同的保证程度。

- *至多处理一次*

  这是最简单的实现，而且适合用在可以接受消息丢失的应用上。使用这种水平的处理时，我们保证了一个元组被处理的次数永远都不会超过一次，但是如果处理过程中出现了问题，元组可能没经过处理就被舍弃。比如，出于提醒的目的，到达的事件要执行一些简单的聚合。

- *至少处理一次*

  这个水平保证每个元组至少成功处理一次，但是也可以接受元组被延迟而且被处理多次。比如，如果在确定元组处理之前任务崩溃，Storm 重新处理元组，那么元组可能会被重复处理。增加这种保证水平只需要在代码中增加实现拓扑的内容，补充的代码相对较少。举一个符合这种情况的例子：有一个处理信用卡的系统要确保在出现错误时重新获得授权。

- *仅处理一次*

  这是最复杂的一个保证方式，元组需要处理一次，而且只能是一次。这种保证水平能用于要求幂等性处理的应用——换句话说，处理同一个元组的组总是需要得到相同的结果。注意，这种保证水平可能会影响核心 Storm 之上的其他抽象，比如 Trident。因为这需要特定的处理，我们将在 7.2.7 节进一步讨论只处理一次要求的内容。

注意，有了可靠的消息来源，消息处理功能才能得到保证。也就是说，如果元组处理过程中出现错误，一个 Storm 拓扑的 spout 与消息来源需要有能力重新发送消息。Kafka 经常与 Storm 一起使用，原因是在出现错误时它能够让 spout 请求再次发送消息。

7.2.7 仅处理一次机制

在要求仅处理一次时，Storm 提供了两个选择。

- *事务性拓扑*

  在一个事务性拓扑（transactional topology，http://storm.apache.org/releases/current/Transactional-topologies.html）中，元组处理基本上被分成两个阶段。

  - 处理阶段，并行处理批元组。
  - 提交阶段，保证按照严格的顺序提交批次。

这两个阶段包含一个交易，能够并行批处理，保证批次提交并在需要时重新发送。最近几个版本的 Storm 都弃用了事务性拓扑，而是使用 Trident，我们随后会进行介绍。

- Trident

  与事务性拓扑相似，Trident（http://storm.apache.org/releases/current/Trident-tutorial.html）是 Storm 上的一种高水平抽象，提供了多种优势。

- Trident 提供熟悉的查询操作符，如合并、聚合、分组、函数等。使用 Pig 的开发者会十分熟悉项目模型。
- 重要的一点是，Trident 支持仅处理一次语义。

正如你所注意到的，目前 Trident 更为适合实现仅处理一次的情况。由于 Trident 的代码与执行操作与 Storm 相差很大，所以本章中后面的内容会仔细地梳理 Trident。对于项目模型，Trident 与 Spark Streaming 非常相似，我们随后也会看到。

7.2.8 容错性

除了能够保证元组处理，Storm 也能在处理过程出现偶然情况或硬件出现错误时（http://storm.apache.org/releases/current/Fault-tolerance.html）保证处理。Hadoop 用户会非常熟悉以下功能。

- 如果 Worker 停止，Supervisor 处理会将其重启。持续出现错误时，任务会转移到另一台机器。
- 如果服务器停止，分配给那个主机的任务会超时，因而会重新分配给其他主机。
- 停止的 Nimbus 与 Supervisor 处理可以重新启动，而且不会影响 Storm 集群上的数据处理。另外，如果 Nimbus 处理速度降低，Worker 能够继续处理。

7.2.9 Storm与HDFS集成

之前我们就注意到，Storm 实际上是单独的数据处理系统，但是很容易与外部系统，以及数据来源和数据 Sink 集成。现在 Storm 项目中包含一种 HDFS bolt（https://github.com/apache/storm/tree/master/external/storm-hdfs），能够将文本与序列文件写入 HDFS。但是也应该注意，该 bolt 与 Hadoop 的集成相对基础，不能像 Flume HDFS Sink（https://flume.apache.org/FlumeUserGuide.html#hdfs-sink）那样支持更加丰富的功能。

在架构中使用 HDFS bolt 需要注意以下几点。

- 在写入时，HDFS bolt 的实现支持一次写入一个元组，同时能够根据处理的元组数量指定同步文件输入到 HDFS 中的频率。但是，这种方法可能会带来数据丢失，比如，在获取元组之后，bolt 就停止了，但此时还没有同步到 HDFS。
- 同样也是在写入时，HDFS bolt 不支持纯文本或者序列文件之外的格式（如 Avro）。

值得注意的是，这些并不是 Storm 框架本身的限制，随着 HDFS 集成的不断发展，问题很可能会得到解决。

7.2.10 Storm与HBase集成

我们讨论了 Storm 与 HDFS 集成时需要考虑的问题，这些问题同样适用于 Storm 与 HBase（https://github.com/ptgoetz/storm-hbase）的集成。对于 HDFS bolt，你需要阅读相关文档并检查对应的代码，确保以批量写入的方式将数据写入 HBase 中，而不是一条数据一个 `put`。单条 `put` 操作每执行一次，WAL（Write-Ahead Log）日志都会同步写到 HDFS 中三个不同节点的磁盘上。这样会对 HBase 写入造成很大的性能影响。表 7-1 为在一个小的测试集群

上简单测试的性能。这些结果与你期望在生产集群上看到的并不完全相同，但是能够简单地表现不同批量写入大小的相对性能影响。

表7-1：不同批量写入的相对性能

集群类型	键的字节数	值的字节数	批量写入大小	线程数	每秒的Put次数
3 节点低功耗虚拟机	32	1024	1	4	3180
3 节点低功耗虚拟机	32	1024	10	4	5820
3 节点低功耗虚拟机	32	1024	100	4	38 400
3 节点低功耗虚拟机	32	1024	1000	4	120 000
4 节点实体机	32	1024	1	10	28 533
4 节点实体机	32	1024	1000	10	389 385

回顾 Microbatch 与 Streaming 的对比，这是一个很好的使用案例，Microbatch 有效利用了 HBase 吞吐量。

7.2.11　Storm示例：简单移动平均

在评估 Storm 之前，我们先来看一个例子——即一个小的应用，计算股票报价记录的简单移动平均（Simple Moving Average，SMA）。我们计算了数据流中最后 *N* 个值的平均值，进而实现了一个 SMA，在这里 *N* 是一个特定的周期——计算某值在特定窗口的平均值。

本例给定一个股票价格的数据流，而且窗口大小为 3，输入与输出如下所示。

```
Next value = 33.79, SMA = 33.79
Next value = 35.61, SMA = 34.7
Next value = 35.70, SMA = 35.03
Next value = 35.43, SMA = 35.58
Next value = 34.93, SMA = 35.35
Next value = 34.85, SMA = 35.07
Next value = 34.53, SMA = 34.77
```

窗口大小为 3 时，通常计算数据流中最后 3 个值的平均，舍弃所有的前面的值。主要因为到达的包含值的数据流是移动的窗口，所以这里提到的是*移动平均*（moving average）。

我们在案例中使用了股票价格数据，格式如下。

```
stock symbol, date, opening price, daily high, daily low, closing price, volume,
adjusted close
```

我们将把收盘价格作为 SMA 计算的值。为了做到简洁，数据处理的结果记录将包含在以下 field 中。

```
ticker, SMA
```

除了计算移动平均，本例还使用了 HDFS bolt 将原始记录永久性地保存在 HDFS 中，提供了一个 Storm 拓扑与 Hadoop 的集成示例。本例实现的拓扑如图 7-6 所示。

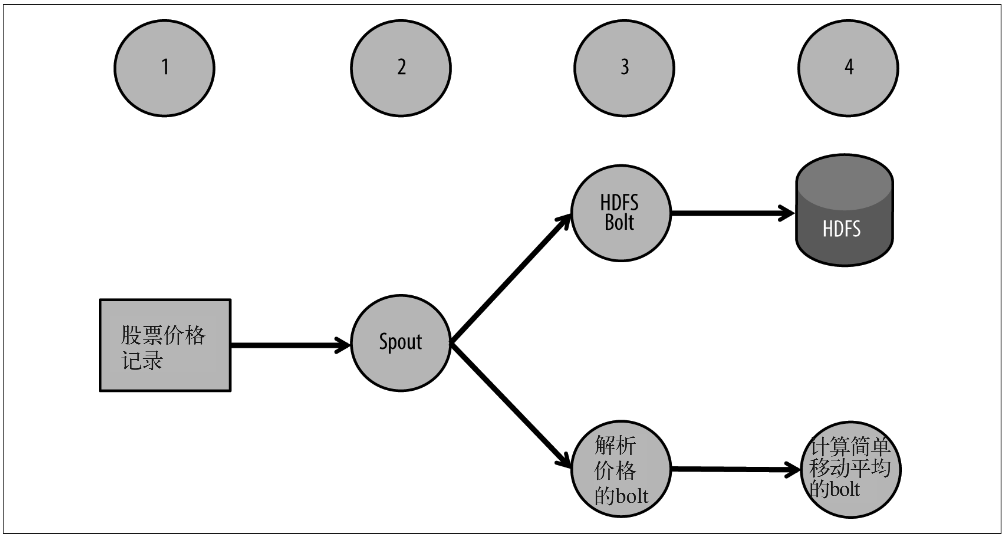


图 7-6：Storm 简单移动平均拓扑

我们来看一下图中的步骤。

(1) 元数据为每天的股票价格记录。为了便于演示，这里假设从本地磁盘上的一个文件中读取记录。实际应用中，记录一般来自可靠的事件来源，如消息队列、Kafka 或 Flume。
(2) 股票价格记录将会通过 Storm spout 发送到拓扑中。
(3) spout 会把元组发送到两个不同的 bolt：一个 HDFS bolt 实例会将原始记录永久性地存储到 HDFS 中，一个解析 field 的 bolt 需要计算移动平均。
(4) 来自解析 bolt 的输出元组会传输到将要计算移动平均的 bolt。

我们看一下实现这个案例所需要的代码。

首先看一下 `StockTicksSpout`，它为理解拓扑提供了一个切入点。为了方便演示，我们将 `StockTicksSpout` 定为一个简单的 spout 实现，只从本地文件读取数据。

```
public class StockTicksSpout extends BaseRichSpout {
  private SpoutOutputCollector outputCollector;
  private List<String> ticks;

  @Override
  public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("tick"));  ❶
  }

  @Override
  public void open(Map map,
                   TopologyContext context,
                   SpoutOutputCollector outputCollector) {
    this.outputCollector = outputCollector;

    try {
```

```
      ticks =
        IOUtils.readLines(ClassLoader.getSystemResourceAsStream(   ❷
        "NASDAQ_daily_prices_A.csv"),
        Charset.defaultCharset().name());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
  }

  @Override
  public void nextTuple() {
    for (String tick : ticks) {
      outputCollector.emit(new Values(tick), tick);  ❸
    }
  }
}
```

❶ 对于该 spout 发送的元组，定义 field 名称。这时我们将每个输入记录都作为元组发送，名称为 `tick`。

❷ 将文件中的每个记录都读取到 `List` 对象中。

❸ 当 Storm 准备好读取下一个元组时，`nextTuple()` 方法将被调用。也要注意，在 `emit()` 方法中，我们给发送的每个元组都分配了一个消息 ID，因此明确地固定了每一个元组。这种情况下，我们简单地将元组本身当作 ID。执行这种固定操作使下游组件能够获知每个元组的处理，从而保证了可靠性。我们将会在随后的 bolt 代码案例中再看一下这种确认操作（http://storm.apache.org/documentation/Guaranteeing-message-processing.html）。

如图 7-6 所示，元组将发送到两个不同的 bolt 进行处理：`ParseTicksBolt` 将从到达的元组中解析需要的 field，HDFS bolt 将到达的元组持久性地存储到 HDFS 中。`ParseTicksBolt` 类如下所示。

```
public class ParseTicksBolt extends BaseRichBolt {

  private OutputCollector outputCollector;

  @Override
  public void prepare(Map config,
                      TopologyContext topologyContext,
                      OutputCollector collector) {
    outputCollector = collector;
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("ticker", "price"));  ❶
  }

  @Override
  public void execute(Tuple tuple) {
    String tick = tuple.getStringByField("tick");  ❷
    String[] parts = tick.split(",");  ❸
    outputCollector.emit(new Values(parts[0], parts[4]));  ❹
```

```
        outputCollector.ack(tuple);  ❺
      }
    }
```

❶ 为该 bolt 发送的元组定义 field 名称。

❷ 从输入数据流中获取价格记录。

❸ 分割每个价格记录。

❹ 发送一个新的元组，该元组由股票代码与收盘价格组成。

❺ 明确固定上游 spout 中的元组，确认该元组的处理。

HDFS bolt 的代码如下。

```
RecordFormat format = new DelimitedRecordFormat()  ❶
  .withFieldDelimiter("|");
SyncPolicy syncPolicy = new CountSyncPolicy(100);  ❷

FileRotationPolicy rotationPolicy =  ❸
  new FileSizeRotationPolicy(5.0f, Units.MB);

FileNameFormat fileNameFormat = new DefaultFileNameFormat()  ❹
  .withPath("stock-ticks/");

HdfsBolt hdfsBolt = new HdfsBolt()  ❺
  .withFsUrl("hdfs://localhost:8020")
  .withFileNameFormat(fileNameFormat)
  .withRecordFormat(format)
  .withRotationPolicy(rotationPolicy)
  .withSyncPolicy(syncPolicy);

return hdfsBolt;
```

❶ 为存储到 HDFS 的数据定义格式。

❷ 每完成 100 元组便同步到 HDFS。

❸ 达到 5 MB 时旋转一次文件。

❹ 将文件存储到 HDFS 中一个称作 stock-ticks 的目录，该目录位于用户的内部目录。

❺ 使用我们刚刚定义的参数创建 bolt。

解析输入元组时，将要执行计算移动平均任务的 bolt 为 `CalcMovingAvgBolt`。

```
public class CalcMovingAvgBolt extends BaseRichBolt {

  private OutputCollector outputCollector;
  private Map<String, LinkedList<Double>> windowMap;  ❶
  private final int period = 5;

  @Override
  public void prepare(Map config,
                      TopologyContext topologyContext,
                      OutputCollector collector) {
```

```
  outputCollector = collector;
  windowMap = new HashMap<String, LinkedList<Double>>();
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {  ❷
}

/**
* 对于输入流中每一个 ticker,计算它的移动平均
*/
@Override
public void execute(Tuple tuple) {
  String ticker = tuple.getStringByField("ticker");  ❸
  String quote = tuple.getStringByField("price");

  Double num = Double.parseDouble(quote);
  LinkedList<Double> window = (LinkedList)getQuotesForTicker(ticker);  ❹
  window.add(num);  ❺

  // 为进行测试,打印至 System.out。实际情况下应将元组
  // 发送至下游bolt或持久层
  System.out.println("----------------------------------------");  ❻
  System.out.println("moving average for ticker " + ticker + "=" +
                    getAvg(window));
  System.out.println("----------------------------------------");
}

/**
* 返回指定ticker的当前窗口。
*/
private LinkedList<Double> getQuotesForTicker(String ticker) {  ❼
  LinkedList<Double> window = windowMap.get(ticker);
  if (window == null) {
    window = new LinkedList<Double>();
    windowMap.put(ticker, window);
  }
  return window;
}

/**
* 返回当前窗口平均值
*/
public double getAvg(LinkedList<Double> window) {  ❽
  if (window.isEmpty()) {
    return 0;
  }

  // 从窗口去掉最早价格:
  if (window.size() > period) {
    window.remove();
  }

  double sum = 0;
```

```
      for (Double price : window) {
        sum += price.doubleValue();
      }

      return sum / window.size();
    }
  }
```

❶ 创建一个 Map，维护每个股票的窗口。正如前面讨论过的，我们可以使用 field 分组来确保一个 `ticker` 中的所有的值都进入同一个 bolt，但是我们不能保证单个的 bolt 只获取单个 `ticker` 的值。我们会使用一个 Map，由股票代码标记，因此我们可以在 bolt 实例中管理多个股票的窗口。

❷ 为了进行测试，我们只打印标准输出，所以不需要声明输出 field。在实际情况中，举例来说，我们可能会将元组发送到一个下游 bolt 或者一个持久层。

❸ 从输入元组中检索股票与价格的值。

❹ 检索与股票相关的值列表（目前的窗口）。

❺ 将目前的值添加到窗口。

❻ 调用方法，计算目前窗口的平均值，并输出结果。

❼ `getQuotesForTicker()` 方法将检索目前窗口所要求的股票。如果窗口不存在，那么创建一个新的窗口。

❽ 将平均值返还到目前的窗口。注意，在计算平均之前我们确保将窗口往前移动。

拓扑配置到 `MovingAvgLocalTopologyRunner` 类的 `buildTopology()` 方法中。这种方法如下所示。

```
private static StormTopology buildTopology() {

  TopologyBuilder builder = new TopologyBuilder();  ❶

  builder.setSpout("stock-ticks-spout", new StockTicksSpout());  ❷

  builder.setBolt("hdfs-persister-bolt", createHdfsBolt())  ❸
    .shuffleGrouping("stock-ticks-spout");

  builder.setBolt("parse-ticks", new ParseTicksBolt())  ❹
    .shuffleGrouping("stock-ticks-spout");

  builder.setBolt("calc-moving-avg", new CalcMovingAvgBolt(), 2)  ❺
    .fieldsGrouping("parse-ticks", new Fields("ticker"));

  return builder.createTopology();  ❻
}
```

❶ 使用 `TopologyBuilder` 类，将 bolt 与 spout 传送到一个拓扑中。

❷ 定义 spout。

❸ 定义 HDFS bolt。

❹ 定义解析股票的 bolt。

❺ 定义实际执行计算移动平均的 bolt。注意，我们使用了 field 分组，以确保一个股票的所有值都会进入到同一个 bolt。

❻ 创建一个拓扑。

7.2.12　Storm评估

我们已经讨论过了 Storm 与 Hadoop 的集成，下面看一下 Storm 是否符合我们在本章开始时定义的其他标准。

1. 支持聚合与窗口

在 Storm 中，类似于计数与计算窗口平均的任务很容易实现，因为现有的 bolt 实现案例能够用于一些常见操作，而且 Storm 中的拓扑与分组功能很好地支持了这种处理类型。

但是，这里也存在两个缺陷：首先，计数器的本地存储与历史信息需要执行持续的计数与移动平均，但并不是容错性的。因此，丢失 bolt 处理时，我们会失去所有的本地持久化的值。这时 Storm 创建的聚合变得不可靠，因此这里需要通过一个批数据任务增加准确性。

第二个问题关乎吞吐量。Storm 通过外部持久化系统（如 HBase 或 Memcached）将计数与平均持久化，进而得到吞吐量。如我们在前面注意到的，单一的 `put` 会带来更多的磁盘同步数据，也需要网络中额外的来回传送。

2. 数据扩充与告警

我们在前面讨论过，Storm bolt 在访问事件时允许有很短的延迟，开发者进而能够实现事件扩充与报警的功能。这在功能上类似于 Flume 拦截器。如果想要将已扩充的数据持久化存储到某一系统（如 HBase 或 HDFS）中，不同于 Storm，Flume 可以在延迟相差不大的前提下，通过某些选项和配置拥有更高的吞吐量。

3. Lamdba架构

注意，并不存在 100% 可靠的数据流方案。复制或丢失数据的情况总是有可能出现。与 Storm 处理结合时，必须确保有精确的数据系统能保证运行批数据任务。实现这种批数据处理的代码通常由 MapReduce 或 Spark 等完成。

7.3　Trident接口

如前所述，Trident 是 Storm 上的高层抽象，能够解决一次处理一个事件时出现的问题。Trident 的设计者并没有创建一个完整的新系统，而是选择把 Storm 包起来，以便支持与 Storm 之间的交互。与 Storm 不同，Trident 后面伴随着一个类似于 SQL 的声明性项目模型。换句话说，Trident 按照是什么而不是怎么做来表达拓扑。

Trident 应用以 Trident spout 替换 Storm spout，但是没有定义 bolt 来处理元组，而是使用 Trident 操作（Trident operation）进行处理。你即将看到，后面的例子没有像 Storm 一样定义明确的拓扑来处理数据流，而是把这些操作串成链，创建了自己的处理流程。Trident 提供了大量操作类型，如过滤、分片、合并与分组。对于使用过 SQL 或者抽象（如 Pig 或

Cascading）的用户来说，大多数的操作类型都不陌生。也就是说，你只能用提供的操作实现 Trident 应用，尽管如此，正如后面的内容会谈到的，这些操作范围之内产生的处理存在一定的灵活性。

当然，Storm 与 Trident 之间还有一个主要差异：Trident 后面会跟随着一个 Microbatch 模型——按照元组的批次而不是单个独立的元组处理数据流。这就提供了一个模型，易于支持只进行一次的语法。

Trident 的一个核心特点是支持可靠的"仅一次处理"，允许从有状态的来源（如内存或 HDFS）读取元组以及写入元组。因此，它在事件出现错误时能够重新发送元组，而且 Trident 能够管理批次，确保元组只处理一次。

7.3.1　Trident示例：简单移动平均

我们将提供一个简单的例子，帮助你了解 Trident 应用。值得注意的是，之前使用的 HDFS bolt 也支持 Trident，尽管这个例子与此无关。

与前面的例子相同，这里要计算股票价格数据的移动平均。与前面的例子不同的是，为了简便行事，到达的数据流将只包含一个单一价格的记录。这些记录的格式如下。

```
date, opening price, daily high, daily low, closing price, volume, adjusted close
```

让我们看一下示例代码中 `MovingAvgLocalTopologyRunner` 的主类。

```
public class MovingAvgLocalTopologyRunner {

  public static void main(String[] args)
      throws Exception {

    Config conf = new Config();
    LocalCluster cluster = new LocalCluster();

    TridentTopology topology = new TridentTopology();

    Stream movingAvgStream =
      topology.newStream("ticks-spout", buildSpout())  ❶
      .each(new Fields("stock-ticks"), new TickParser(), new Fields("price"))  ❷
      .aggregate(new Fields("price"), new CalculateAverage(),
                 new Fields("count"));  ❸

    cluster.submitTopology("moving-avg", conf, topology.build());  ❹
  }
}
```

❶ 创建一个新的 spout，将价格元组批量地发送到拓扑中。这里只是使用一个简单的 spout，从一个本地文件中读取记录并发送包含五个元组的批处理。`buildSpout()` 是一种很实用的方法，在 `FixedBatchSpoutBuilder` 中定义（参见下一个代码区域），这种方法简单地从一个文件中读取股票数据，并且一次发送包含五条记录的批处理。

❷ 对于输入数据流的每个元组，使用 `TickParser` 类解析价格 field。

❸ 使用聚合操作计算平均。我们随后会看到相关代码。

❹ 启动拓扑。

在讲解聚合操作之前，我们先看一下用于 `FixedBatchSpoutBuilder` 工具类的代码。

```
public class FixedBatchSpoutBuilder {
  public static FixedBatchSpout buildSpout() {
    List<Values> ticks = new FixedBatchSpoutBuilder().readData();
    FixedBatchSpout spout = new FixedBatchSpout(new Fields("stock-ticks"), 5,
        ticks.toArray(new Values[ticks.size()]));
    spout.setCycle(false);
    return spout;
  }

  public List<Values> readData() {
    try {
      BufferedReader reader = new BufferedReader(
          new InputStreamReader(getClass().getResourceAsStream(
              "/AAPL_daily_prices.csv")));

      List<Values> ticks = new ArrayList<Values>();
      String line = null;
      while ((line = reader.readLine()) != null) ticks.add(new Values(line));

      return ticks;
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

我们已经注意到，执行计算平均的类是一种聚合操作的实现。这里使用了来自 Trident API 的 `Aggregator`，如下所示。

```
public class CalculateAverage
    extends BaseAggregator <CalculateAverage.AverageState> {

  static class AverageState {
    double sum = 0;
  }

  @Override
  public AverageState init(Object batchId, TridentCollector collector) {  ❶
    return new AverageState();
  }

  @Override
  public void aggregate(AverageState state,
                        TridentTuple tuple,
                        TridentCollector collector) {  ❷
    state.sum = state.sum + Double.valueOf((Double)tuple.getValue(0));
  }

  @Override
  public void complete(AverageState state, TridentCollector collector) {  ❸
    collector.emit(new Values(state.sum/5));
  }
}
```

❶ 在每个批次被处理之前调用 `init()` 方法。这种情况下，我们只创建一个包含状态的新对象。在这里，状态只是批次中收盘价格的简单加和。

❷ 批次中的每个元组都调用 `aggregate()` 方法，而且该方法会简单地更新收盘价格的总和。

❸ 当批次中所有的元组都经过处理后，调用 `complete()` 方法，并将计算的得出的平均发送到批次。

与 Storm 中的例子相比，这个例子有些简单，但 Trident 代码明显比 Storm 代码更精确。我们没有通过编写多行代码定义一个拓扑，只是简单定义操作来执行处理。使用的代码如下。

```
Stream movingAvgStream =
  topology.newStream("ticks-spout", buildSpout())
  .each(new Fields("stock-ticks"), new TickParser(), new Fields("price"))
  .aggregate(new Fields("price"), new CalculateAverage(), new Fields("count"));
```

我们需要编写一些自定义代码来计算移动平均，但是 Trident 也包含很多内置的操作，能够实现常见的处理任务。

我们已经概述了 Trident，如欲了解更多，请参阅在线文档（http://storm.apache.org/releases/current/Concepts.html）或 Storm Applied。

7.3.2　Trident评估

了解了 Trident 的作用方式，下面我们来看一下它是如何与数据流评估标准相联系的。

1. 支持计数与窗口化

与核心 Storm 相比，Trident 有很大的提高。借此，我们能够持久化到外部存储系统，维持较高的吞吐量。

2. 数据扩充与告警

这里应该有一些与核心 Storm 相同的功能。对于 Trident，批次只是一层封装。这与 Spark Streaming 中的批不同，后者是真正的 Microbatch，表明在发送第一个事件之前存在延迟。相反，Trident 中的“batch”只不过是一组元组结尾处的标记。

3. Lamdba架构

再说一次，并不存在完全可靠的数据流方法，但是在这里 Trident 要优于 Storm。尽管为了支持 Lamdba 架构，我们仍然需要在 MapReduce 或 Spark 等应用中实现批处理。

7.4　Spark Streaming

Spark Streaming 是数据流处理领域的新事物。Spark Streaming 充分利用了 Spark RDD 的强大之处，并与可靠性及只进行一次处理的语义结合。与 Trident 类似，Spark Streaming 提供了一个数据流处理方案，其吞吐量非常高。

如果你的使用案例允许在 2~5 秒的 Microbatch 中处理数据，那么使用 Spark Streaming 会带来以下优势。

- 可靠的中间数据持久化，用于计数与移动平均。

- 支持与外部存储系统（如 HBase）集成。
- 在数据流与批处理之间复用代码。
- Spark Streaming 的 Microbatch 模型允许处理帮助降低复制事件的风险的模式。

7.4.1 Spark Streaming概述

已经熟悉 Spark 的用户会发现，Spark Streaming 在代码风格与 API 方面同 Spark 非常相似。我们首先探讨 Spark Streaming 的 `StreamContext`，它是 `SparkContext` 的封装。回忆一下，`SparkContext` 是标准 Spark 应用的根本。`StreamContext` 最大的不同之处在于一个被称作 `batchDuration` 的参数。这个周期会设定目标批次间隔，我们稍后会对此进行解释。

核心 Spark 使用 `SparkContext` 创建初始 RDD。与此相同，Spark Streaming 使用 `StreamingContext` 创建初始的 `DStream`。`DStream` 是 RDD 的封装。两者的不同之处在于它们实际上代表不同的东西。标准 RDD 引用了一个分布式的、不可变的集合，而 `DStream` 引用了与批次窗口相关的、分布式的、不可变集合。为了了解一下背景，我们可以把一个 `DStream` 看作一个电视，屏幕上出现的景象在不断地改变，但是背后是静止的图像。本文中，RDD 是不会改变（不可变的）的图像，而 `DStream` 是电视，看起来可变，实际上由一系列不可变的 RDD 组成。

这个思路比较复杂，所以我们来看 Spark Streaming 的几个例子，然后再用图例的方式进行解释。

7.4.2 Spark Streaming示例：简单求和

我们首先讲一个例子。这个例子使用 `StreamingContext` 创建了一个 `InputDStream`，而后执行一个简单的过滤与求和，如图 7-7 所示。

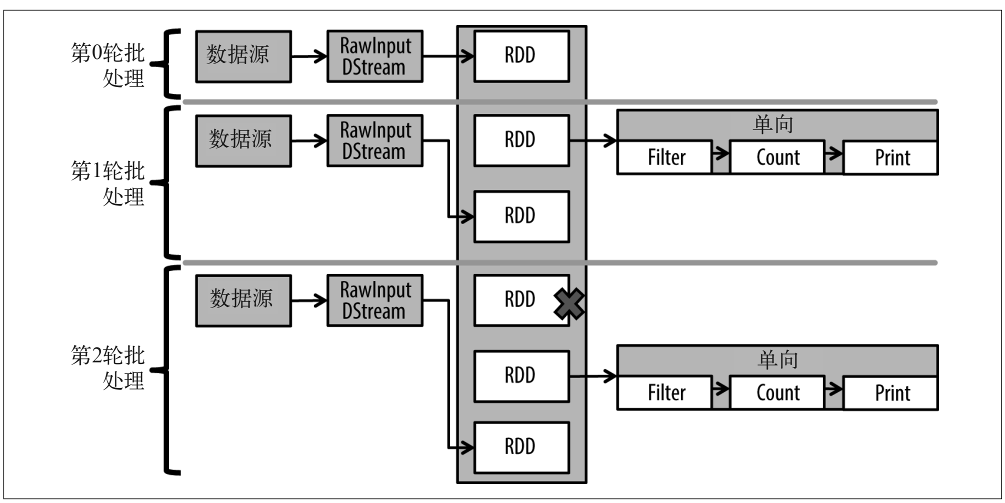


图 7-7：Spark Streaming 简单求和

深入研究一下这张图，我们可以总结出以下几点。

- 每一点都是以批为基础的，而且该图展示了同一系统不同时间点的情况。
- InputDStream 通常会收集数据并将数据放置到一个新的 RDD 中处理。
- 执行 DStream 时不会再获取数据。执行状态时它是一个不可改变的 DStream。我们随后会讨论相关内容，这保证了批次可以成功地完成重新发送。
- Spark 在同一引擎上运行。
- 在 DStream 的上下文中，RDD 在不需要的时候移除。从图上看，似乎只要执行下一步，RDD 就被移除，但这其实是工作方式的过度简化。移除的主要目的是清除未使用的旧 RDD。

后台代码如下所示。这个例子对每秒发送到 Spark Streaming 中包含单词 the 的句子计数。

```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream = ssc.socketTextStream(host, port, StorageLevel.MEMORY_ONLY_SER_2)
rawStream.filter(_.contains("the")).count().print()
ssc.start()
```

7.4.3　Spark Streaming示例：多路输入

现在，我们往前走一步，增加更多的 InputDStream，再看看会是什么样子。在图 7-8 中，我们拥有三个 InputDStream，而且在执行过滤与求和之前将其联合。这里展示了处理获取的多路输入流，以及一个联合的执行。我们将其当作一个单一的数据流。注意，在联合之前执行过滤与求和会更高效，但是我们在这个例子中要尽量少做改变，以便阐明多路输入的含义。

该处理如图 7-8 所示。

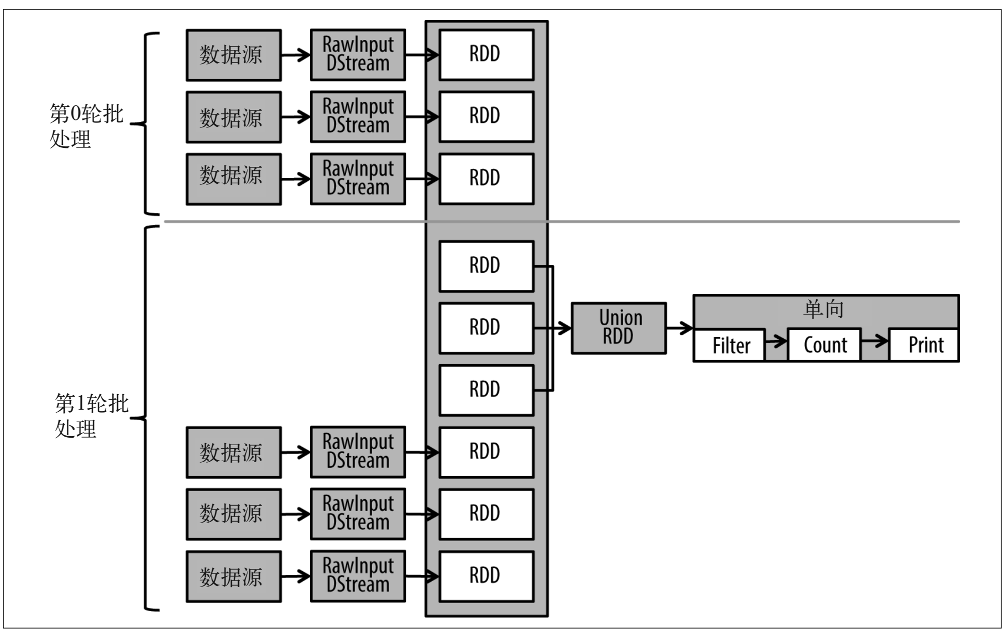


图 7-8：Spark Streaming 多路输入

代码如下。

```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream1 =
  ssc.socketTextStream(host1, port1, StorageLevel.MEMORY_ONLY_SER_2)
val rawStream2 =
  ssc.socketTextStream(host2, port2, StorageLevel.MEMORY_ONLY_SER_2)
val rawStream = rawStream1.union(rawStream2)
rawStream.filter(_.contains("the")).count().print()
ssc.start()
```

其中并没有多少新东西，我们只是增加了另一个输入，这个输入能够增加更多数据流。我们可以选择进行联合，而且在一个单一的 Microbatch 中执行任务。但是这不表示合并数据流都是有意义的。在处理大容量数据时，最好永远都不要合并数据流，以免增加延迟。

很多设计模式都可以对到达的数据分区，比如输入数据流来自不同 Kafka 分区的情况。注意，通过分区进行联合或合并代价都比较大，所以如果应用不要求使用就应该避免。

7.4.4　Spark Streaming示例：状态维护

下一个例子解释了在间隔之间（比如执行连续求和的情况）维护状态转移的思路，如图 7-9 所示。

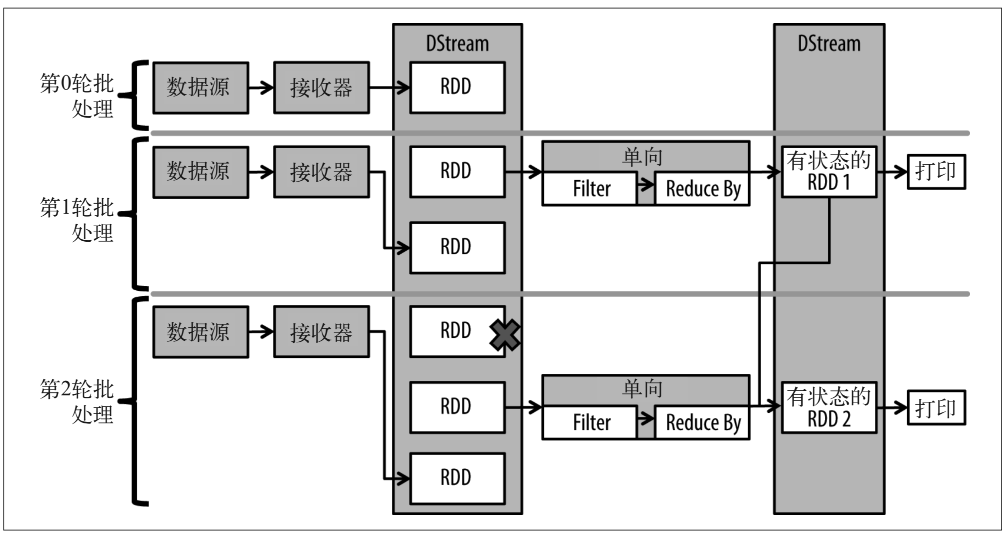


图 7-9：Spark Streaming 中的状态维护

代码如下。

```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))
val rawStream = ssc.socketTextStream(host, port, StorageLevel.MEMORY_ONLY_SER_2)

val countRDD = rawStream.filter(_.contains("the"))
  .count()
```

```
  .map(r => ("foo", r))

countRDD.updateStateByKey((a: Seq[Long], b: Option[Long]) => {  ❶
  var newCount = b.getOrElse(0l)
  a.foreach( i => newCount += i)
  Some(newCount)  ❷
  }).print()

ssc.checkpoint(checkpointDir)  ❸

ssc.start()
```

这是 `DStream` 交叉进入批处理的第一个例子。在这里，一个批次的总和被用于合并新的总和，然后产生一个包含新总和的新 `DStream`。

下面深入研究这段代码。

❶ 首先要注意的是，用于 updateStateByKey() 的函数基于一个键，所以只会将值传入，尽管值的传入方式是 Spark 独有的。第一个值是 Seq[Long]，来自 Microbatch 的这个键包含了所有可能的新值。该批次很可能不包含任何一个值，而且这会是一个空的 Seq。第二个值是一个 Option[Long]，代表来自最后一个 Microbatch 的值。因为可能不存在值，所以 Option 允许它为空。

❷ 注意，这里返回的是一个 Option[Long]。因此我们能够使用一个 Scala None，选择移除填充在先前 Microbatch 中的值。

❸ 最后要注意的是，我们已经增加了一个检查点文件夹（checkpoint folder）。这有助于重新存储 RDD。这些 RDD 穿过 Microbatch，并且能够提供额外的数据丢失防护。

DStream 的不可变性是如何提供容错的

对于出现错误时 Spark Streaming 执行恢复的方式来说，这是一个很好的讨论切入点。在目前涉及的例子当中，我们已经看到，`DStream` 总是由很多不可改变的 RDD 组成。也就是说，它们创建之后不能改变。

在 `updateStateByKey()` 命令中有状态的 `DStream` 就是一个极好的例子。合并批次 5 中的结果与前面批次的有状态 `DStream` RDD，就创建了新的有状态的 `DStream`。

出现错误时，Spark 有两种选择：对于每 *N* 个 Microbatch，有状态的 `DStream` 传输到一个检查点目录，*N* 可配置。因此，配置在每个 Microbatch 上进行写入，我们就能够从检查点目录中恢复。如果将 *N* 设定为一个大于每个批次的值，那么我们需要使用用来创建它的 RDD 重新创建状态。

这里需要在磁盘 `I/O` 上作出牺牲，但在实践中，分布式情况决定这会是一个比较小的牺牲。你也可以使用额外的内存来存储重新创建的路径。注意，选择制图版恢复错误更有效，但在本文写作时这还是状态的东西。

注意，这种有状态的 `DStream` 与 Trident 状态相似，但它们之间也存在不同。它仍然在 Spark Streaming 中，而且是容错的。Trident 需要你在一个外部持久性系统上重新启动，否则无法获得同样的功能并提供容错。而 Spark Streaming 中的 `DStream` 是容错的。

7.4.5 Spark Streaming示例：窗口函数

现在，假设我们想对一段时间进行滚动求和。实现的方式如以下代码所示。

```
val sc = new SparkContext(sparkConf)

val ssc = new StreamingContext(sc, Seconds(2))
val rawStream = ssc.socketTextStream(host, port, StorageLevel.MEMORY_ONLY_SER_2)

val words = rawStream.flatMap(_.split(' '))
val windowCount = words.countByValueAndWindow(Seconds(20), Seconds(6))
windowCount.count.print()

ssc.checkpoint(checkpointDir)

ssc.start()
```

那么这又是怎样工作的呢？如图 7-10 所示，Spark Streaming 将存储一束 `DStream`，而且使用这些 `DStream` 创建第一个结果，直到达到第六个 pass。在第六个 pass 上，我们将增加最后一个结果与新值，并且移除最旧的值。由此，在执行时间固定时，该 pass 会延展到 20，30，100 的窗口。

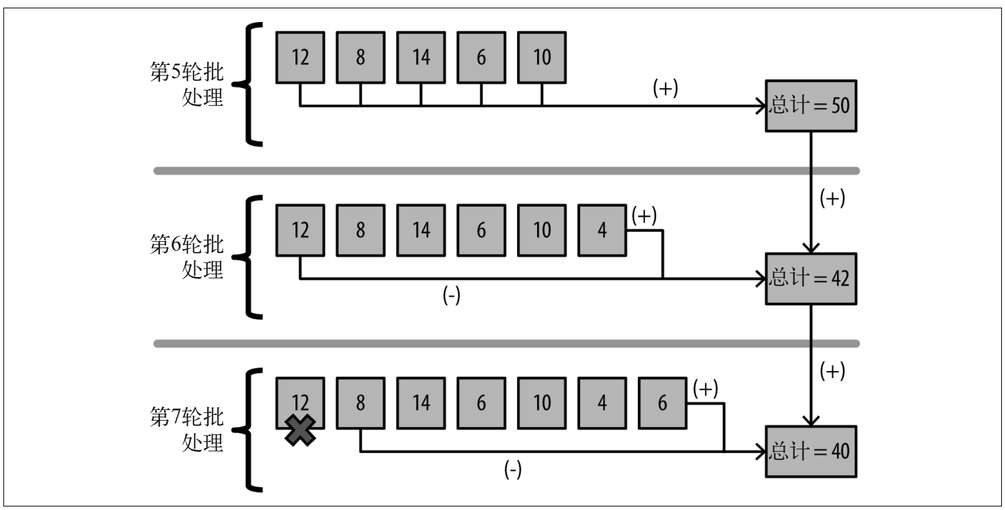


图 7-10：Spark Streaming 窗口示例

7.4.6 Spark Streaming示例：Streaming与ETL代码比较

在随后的例子当中，我们会看一下 Spark Streaming 与 Spark 之间的代码差异。再次使用词汇计数作为例子，假设我们想在一个数据流模式，以及一个批量模式中运行程序。下面看一下 Spark Streaming 与 Spark 的代码以及执行时的差异。

Spark Streaming 的代码如下。

```
val lines = ssc.textFileStream(args(0))
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
```

Spark 的代码如下。

```
val lines = ssc.textFileStream(args(0))
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
```

Spark 与 Spark Streaming 的项目模型是一样的，以上代码也证明了这种说法。如同刚才所说的，某些情况下代码与流程在批次中与数据流中的执行会有所不同。但是，即使在那种情况下，大多数代码也都在转化函数中。

以下例子有所不同。前面的代码片段都执行同样的任务，一个用于一个批次而另一个用于一个固定的输入事件集。但是如果我们想要比较一个连续的词汇求和呢？那么我们可以看一下后面的代码。

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).updateStateByKey[Int](_ + _)
wordCounts.print()
```

尽管代码并不完全相同，但背后的引擎是相同的。代码本身含有的差异较小，而且仍然能利用函数复用。

对于 Spark Streaming 的功能，我们的讨论还不够多，但是希望前面的例子能够为你提供基本的理解。

要进一步研究 Spark Streaming，请查看 Spark 文档（https://spark.apache.org/docs/latest/streaming-programming-guide.html）或者查阅 *Learning Spark*（http://shop.oreilly.com/product/0636920028512.do）。

也要注意，每发布一个版本，开放给 Spark Streaming 的 Spark 的功能就越多。这也意味着，类似于 GraphX 与 MLlib 的工具能够成为 NRT 方法的一部分，为支持不同数据流的使用案例提供更多机会。

Spark Streaming 容错

Spark Streaming 将预写式日志（Write Ahead Log，WAL）用于驱动器处理，与 HBase 相似，所以在驱动器处理出现错误时，另一个驱动器处理能够启动，而且能够将状态从 WAL 中集结起来。使用了 WAL，Spark Streaming 中的数据不可能丢失，因为 DStream 在 RDD 上是内置的，而 RDD 为弹性设计。弹性程度由开发者确定，这一点与标准的 Spark 任务相同。

因为出现故障时数据得到了“保护”，所以故障时 Spark Streaming 会知道需要重新运行哪个转换或操作。Microbatch 系统只需要跟踪每个批次的成功或失败情况，而不用关注每个事件。从性能上讲，这是一个很大的优势。事件级别的监控会对类似于 Storm 的系统造成很大的影响。

7.4.7 Spark Streaming评估

现在我们已经基本理解了 Spark Streaming 的工作流程，所以下面总结一下 Spark Streaming 的功能与工作方式。

1. 支持求和与窗口

求和、窗口与滚动平均在 Spark Streaming 中非常简单。与其他系统（如 Storm）相比存在一些差异。

- 进行求和或者聚合增量时，平均延迟超过 0.5 毫秒。
- 可以从一次 Worker 错误中恢复，求和更可靠。
- 对于 HBase 之类的系统，从持久求和与平均来说，Microbatch 将有益于吞吐量，因为 Microbatch 允许批次 `put` 与增量。
- 与 Microbatch 相关的回复也将帮助减少由 Spark Streaming 创建的副本。

2. 数据扩充与告警

在数据扩充与报警方面，Spark Streaming 能够按照 Flume 与 Storm 的方式实现。如果要求从外部系统（如 HBase）查找然后执行数据扩充与报警，那么 Spark Streaming 甚至可能会存在性能和吞吐量的优势。

这里的主要缺陷是延迟。我们之前说过，Spark Streaming 是 Microbatch，所以如果要求在 500 毫秒内报警，那么就不应该选择这个工具。

3. Lambda架构

是的，你猜到了：并不存在完美的数据流方法。尽管 Spark Streaming 不完美，但它提供了一些优势，如代码复用。用于 Spark Streaming 的代码与 Spark 中实现 ETL 的代码非常相似。这将降低维护成本，化解新兴企业在财务上的烦恼。

7.5 Flume拦截器

本书的其他内容已经深入介绍了 Flume，所以这里不会再详细介绍。本章内容只包括 Flume 的数据源与拦截器组件，如图 7-11 所示。

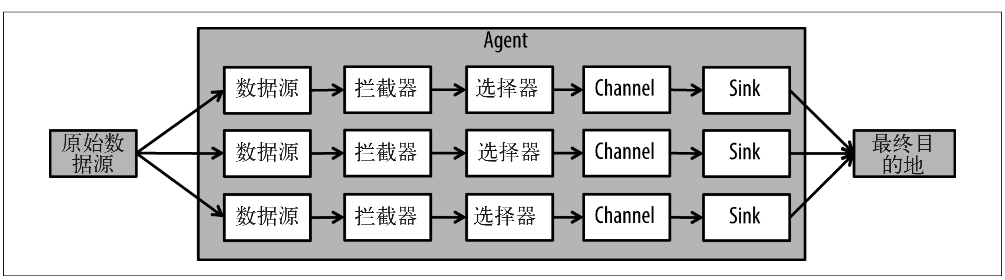


图 7-11：用于数据流处理的 Flume 拦截器

拦截器在 Channel 前面，所以事件只要收集到就会被处理。或者，一个批次中预定数量的事件有了定义，事件也会被处理。

在选择 Flume 而不选择其他方法时，本章重点考虑到，Flume 非常适合采集数据。验证或报警的情况通常伴随一个持久化要求，所以这一点非常重要。欺诈报警就是一个例子，为了以后进行检查，一般要求每个事件与事件的报警决策都存储到 HDFS 中。

7.6 工具选择

我们已经在这里讨论了很多种方法以及这些方法的优势与缺陷。注意，只要找到了适合的使用情况，每一种系统都是有优势的。这些系统都是不同的引擎，优化与重点稍有差异。你要根据自己的使用要求选择合适的工具，这里没有一劳永逸、一成不变的选择教条。我们将讨论几个案例场景，并探讨使用这些工具解决问题的不同方法。

我们将关注三个不同的使用案例。

(1) 采集之前，事件的低延迟扩充、验证与报警。
(2) NRT 求和、滚动平均与迭代处理，如机器学习。
(3) 更复杂的数据流水线。基本上包括前面提到的所有一切：扩充、验证、报警、求和与滚动平均，以及最后的采集。

7.6.1 低延迟的数据扩充、验证、报警及采集

对于这个任务，我们将通过两个选择来探究不同的解决方法。两种设计都被用于实际应用中而且每个方法都有优势。举例来说，在欺诈检测时你可能需要这种功能，我们将在第 9 章中详细讨论。

方法1：Flume

图 7-12 为一个实现这种功能的 Flume 工作流。如图 7-12 所示，这是一个相当简单的工作流，由一个自定义拦截器与一个 HDFS Sink 组成。Flume Channel 提供了一种数据保护的自定义水平，我们先前在介绍 Flume 时已经提到过。拦截器在 Channel 前面，因此在处理事件时也会稍有延迟。

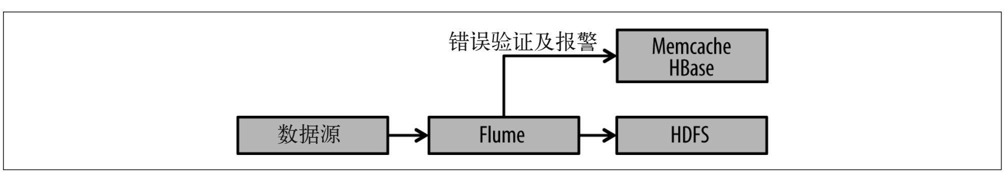


图 7-12：采用 Flume 的使用案例

方法2：Kafka与Storm

第二种方法如图 7-13 所示，即 Storm 与 Kafka 联合，使用 Storm 处理，使用 Kafka 进行事件传输。Kafka 提供了可靠的事件传输，而 Storm 提供了可靠的事件处理。

注意，我们使用 Kafka 并不只是为了将事件传输到 Storm，也是为了将事件采集到 HDFS 中，无论是单独使用还是随意与 Flume 一起使用。我们本来可以使用一个 HDFS bolt 直接从 Storm/Trident 拓扑写入到 HDFS。这样做可能更有效，但是 Kafka 与 Flume 更成熟，而且从 HDFS 集成方面来看提供的功能更多。

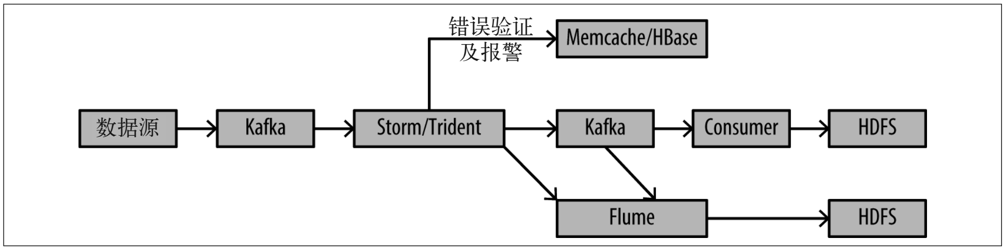


图 7-13：采用 Kafka 与 Storm 的使用案例 1

7.6.2 NRT技术、滚动平均及迭代处理

对于这种情况，并不存在一个很好的低延迟选择。如前所述，Storm 提供了一种很好的低延迟方法，但是要使用批次创建一个巨大的拓扑非常复杂，这非常困难。即使我们克服了前期困难，与 Spark Streaming 或 Trident 相比，后续维持修改也会非常棘手。

我们可以在 Storm、Trident 或 Spark Streaming 中创建这种使用案例。每种方法如图 7-14 所示。

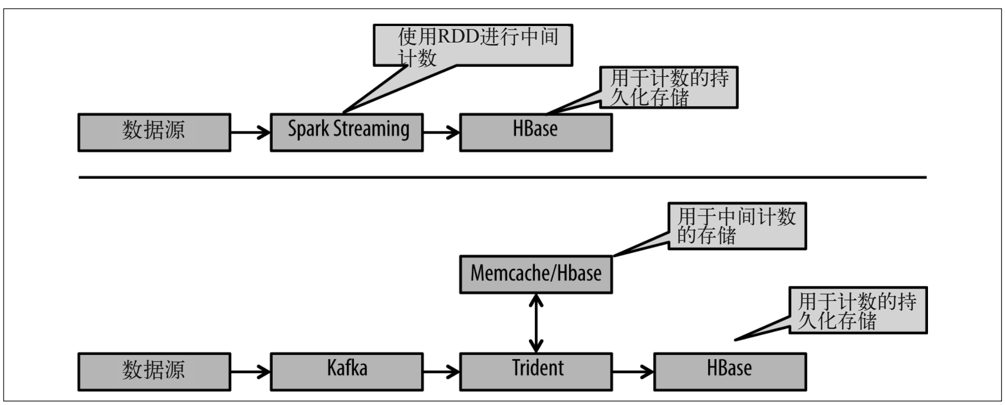


图 7-14：在 Trident 与 Spark Streaming 中创建的使用案例 2

在决定选择哪种方法时要考虑以下因素。

- *批次与数据流的项目模型相同*

  如果你很在意批次与数据流的项目模型是否相同，那么我们强烈推荐你使用 Spark Streaming。这样一来，数据流与处理就能够使用相同的语言与模型，这为培训、管理与维护提供了优势。

- *每个节点的吞吐量*

  这样或是那样的工具总会有各不相同的评价标准，这没什么好奇怪的。我们不会宣称哪个工具在性能方面更优越。根据使用情况不同，任何一个工具都可能表现出色，这取决于具体情况。所以最好的方法是尝试一个原型，确定一个特定的工具是否能够提供足够的性能。

- 高级功能

  在这里，Spark Streaming 有一定的优势。多亏了背后人数众多的贡献者，它每天都有所进步。

7.6.3 复杂数据流

这种实现需要满足以上所说的所有要求。这里有一个更复杂的数据流，需要我们在前面两个例子中讨论到的所有处理功能。我们即将看到，在这里解决方法只是将前面两个例子的方法加到一起。我们首先看一下 Flume 与 Spark Streaming 方法，如图 7-15 所示。

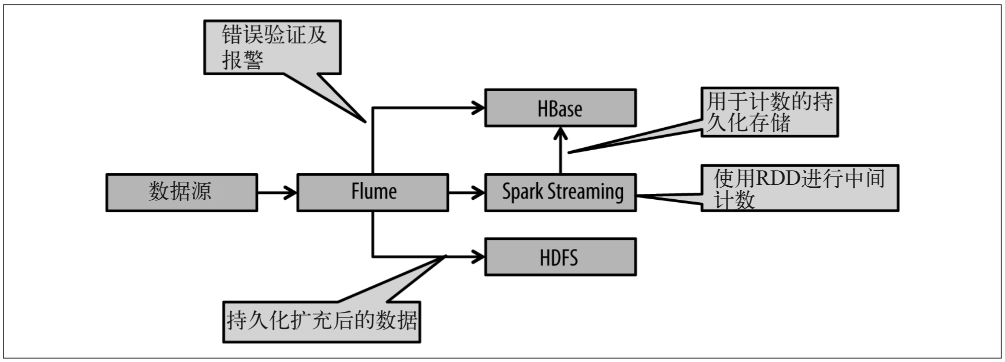


图 7-15：采用 Spark Streaming 的使用案例 3

这里主要是要注意以前未出现的 Flume 与 Spark Streaming 的集成。

现在我们来看一下 Storm/Trident 版本，如图 7-16 所示。

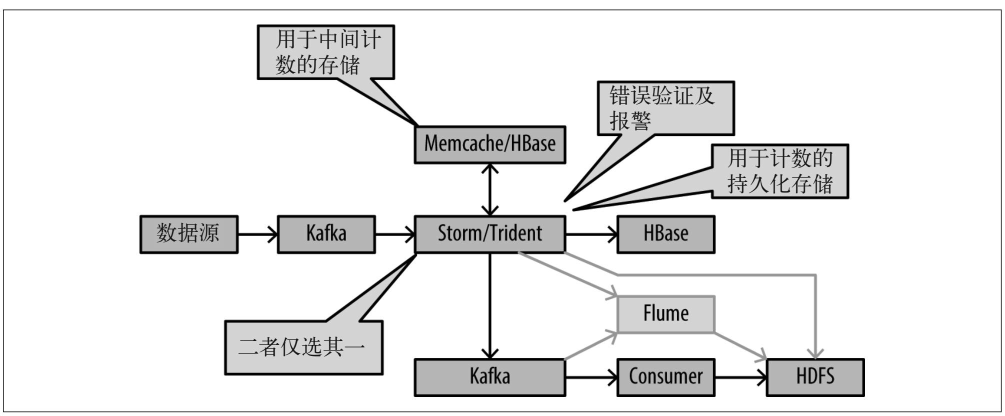


图 7-16：采用 Storm/Trident 的使用案例 3

在这里唯一的更改是，把两个例子的拓扑加到了一起。

7.7 小结

本章再一次提到，Hadoop 正在迅速变成大数据平台，支持多种处理模型。除了能够有效处理批次中的大容量数据，Hadoop 现在还能够作为一个持续处理数据的平台，并与新工具（如 Storm、Trident 和 Spark Streaming）集成使用。我们也探究了经过检验显示可靠的 Flume 怎样才能用于 NRT 任务。支持多种处理模型的功能使 Hadoop 成为了企业数据的枢纽。

我们也在本章中看到，在 Hadoop 上选择工具进行近实时处理时，工具的选取主要取决于使用情况、目前的工具集，以及语言等。选择工具时，你应该全面考虑，而且要确保使用自己的工具执行合适的性能检测，使其符合应用要求。

第二部分

案例研究

第 8 章

点击流分析

点击流分析通常分析用户浏览网站产生的事件，即点击数据。这种分析的目的一般是了解网站访问者的用户行为，从而向基于数据的后续决策提供信息。

点击流分析能够发现用户与线上展示的交互方式、网站流量的地域来源、用户访问网站的高频设备和操作系统、用户完成特定行为（这里一般指某种目标，比如订阅邮件列表、注册新账户、加入购物车）的通常路径等。有了点击数据和市场推广信息，你便可以进行进一步的分析，比如分析各个营销渠道（自然检索、付费搜索、社交媒体支出等）的投资回报率（Return On Investment，ROI）。另外，还可以将点击数据与操作数据存储层（Operational Data Store，ODS）或客户关系管理系统（Customer Relationship Management，CRM）数据相关联，基于更丰富的用户信息深入研究。

本章以点击流数据为例，讲解如何将第一部分中的各种工具结合在一起，但是本章涉及的各个概念适用于任何对机器产生的数据进行批处理的系统。此类数据包括但不仅限于以下几种：广告展示数据、性能或其他日志，还有网络日志及通信日志。

8.1　用例场景定义

假设你是出售自行车部件的网络零售商。用户可以访问你的网站，浏览自行车的零部件、浏览评论，还可以下单付款。你的网站程序会将用户的页面浏览和链接点击全部记录到日志中。一般情况下，网站产生的是纯文本的 Apache 或 Nginx 日志，有些时候点击日志是网络分析工具通过跟踪网站形成的定制化的日志。但无论怎样，点击日志会严格按照时间戳进行追加。

作为网络零售商，你肯定想更加了解自己的用户群，想进行营销和市场方面的优化。为了达成这样的目标，你需要回答如下问题。

- 上个月我的网站上有多少页面浏览量（Page View，PV）？环比增长率如何？
- 上个月我的网站上有多少独立访客数（Unique Visitor，UV）？环比增长率如何？
- 访客购物时的平均停留时间有多长？
- 网站的跳出率（bounce rate）是多少？换句话说，有多少用户打开本网站后尚未跳转至其他页面便已结束访问？
- 对于每个页面，新用户登录并在会话中发生购买行为的概率是多少？
- 对于每个页面，新用户登录并在七天内发生购买行为的概率是多少？
- 购买行为是通过哪一种营销渠道（直接访问、自然检索、付费搜索等）发生的？

想要回答以上乃至更多的问题，则需扫描、处理和分析 Web 服务器记录的日志。

为便于后续讨论，我们假定网络日志均以最常见的格式出现，即*组合日志格式*（combined log format）。此类格式中的每一条日志皆如下所示（注意：限于排版和篇幅，如下示例会在多行呈现，但实际情况下，一条日志记录即为连续的一行）。

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /
apache_pb.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html"
"Mozilla/4.08 [en] (Win98; I  ;Nav)"
```

以上日志具体包含以下部分。

- `127.0.0.1`

  此处是源 IP 地址（即发生点击的设备的 IP 地址）。

- `-`

  第二个字段在例子中使用连字符表示，它一般指客户端标识。实际情况下我们通常认为这不可信，而且会直接被服务端忽略。

- `frank`

  如果需要的话，在此处写明用户名，即 HTTP 认证的身份标识。

- `10/Oct/2000:13:55:36 -0700`

  该字段记录服务完成处理请求的时间，并带有时区。

- `GET /apache_pb.gif HTTP/1.0`

  该字段表示请求的类型，以及客户端请求的页面。

- `200`

  此处是 Web 服务器返回给客户端的状态标识码。

- `2326`

  该字段表示返回给客户端的对象大小，不包括响应报文头部的大小。

- `http://www.example.com/start.html`

  如果需要的话，用该字段代表引用地址（若存在），即引导用户进入当前地址的上一地址。

- `Mozilla/4.08 [en] (Win98; I  ;Nav)`

  本字段是用户代理字符串，能够标识发生点击行为的设备具有的浏览器和操作系统。

8.2 使用Hadoop进行点击流分析

一个活跃的站点每天能够产生数 GB 的数据。存储和分析如此庞大的数据需要一个十分健壮的分布式系统。另外，日志数据的生成是非常迅速的。通常情况下，日志在一天甚至一个小时之内需要多次轮转更替。这是因为分析人员想要尽早知道最新的变化与总体的趋势。此外，日志数据是半结构化的，一条日志记录可能有需要解析的用户代理字符串，也可能有新增或者不久便会移除的字段，如测试用的参数。考虑到这些特点，Hadoop 仍不失为一个点击流分析的利器。

本章会展示如何使用 Apache Hadoop 及生态圈中相关的其他项目，构建一个能够对点击流数据进行收集、处理和分析的应用。

8.3 设计概述

我们将此案例的整体架构分为五大部分：数据存储、数据采集、数据处理、数据分析及协调调度。

- 数据存储模块对存储系统（如 HDFS 或 HBase）、数据格式、压缩算法以及数据模型等进行技术选型。
- 数据采集模块从网络服务器或二级数据源（如 CRM、ODS、市场推广数据）等获取点击流数据，并将其加载到 Hadoop 中待处理。
- 数据处理模块将原始数据导入 Hadoop，并对其进行转换加工，形成支持分析和查询的数据集。此处的处理一词包括但不仅限于以下内容：数据去重、去除无效点击、将点击数据转化为列式存储格式、生成会话（通过关联每次点击中代表这次访问的会话 ID，将单个用户单次访问的多个点击过程合并）及聚合等。
- 数据分析模块在预处理后的数据集上进行各种分析类的查询，以找到本章前面提到的问题的答案。
- 协调调度模块自动组织和协调进行数据采集、处理、分析的各个过程。

图 8-1 展示了这一设计的整体规划。

我们的设计实现将采用 HDFS 作为数据存储，使用 Flume 收集 Apache 日志，使用 Sqoop 将其他的二级数据源（如 CRM、ODS、市场推广数据）导入 Hadoop。我们还将采用 Spark 进行预处理，采用 Impala 分析处理后的数据。将 BI 工具连接到 Impala 之后，我们就能够在这些数据上进行交互式的查询。我们还将使用 Oozie 协调调度多个操作，使其成为一个完整的工作流。

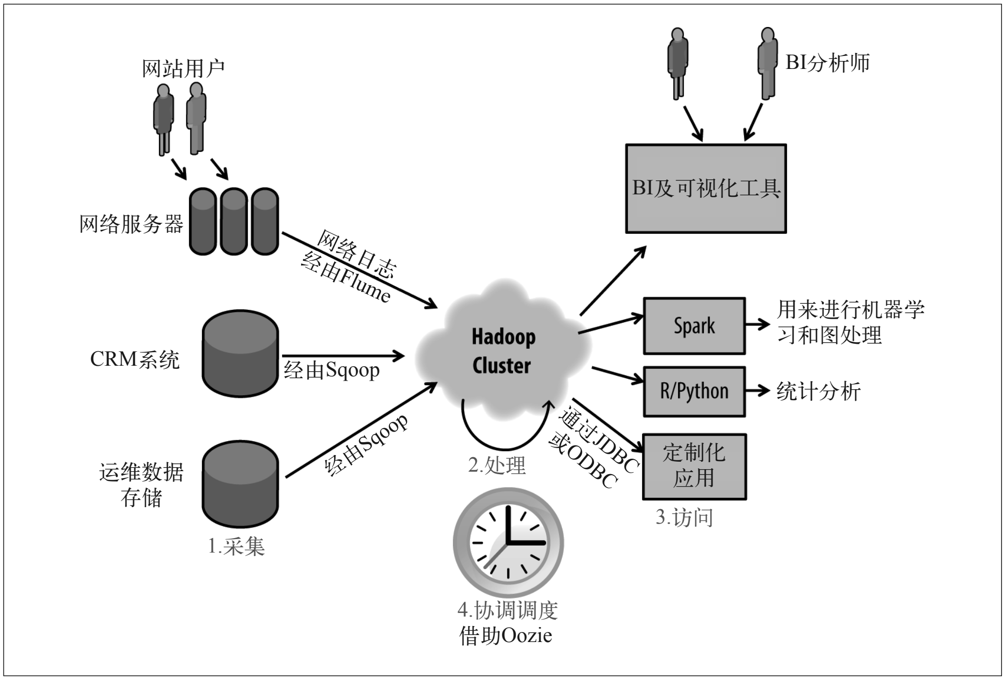


图 8-1：设计概览

接下来介绍设计方案的具体内容，包括文件格式与数据模型的选型、使用 Web 应用将点击流信息发送到 Flume 中、配置 Flume 将这些数据存储成文本格式等细节。我们还会讨论这里涉及的各种处理算法（尤其是会话生成算法及其实现方式），如何使用 Impala（或其他 SQL-on-Hadoop 工具）在海量数据集上进行高效的聚合操作。我们也将在合适的地方讲解数据的自动处理（如数据采集、数据加工等）。另外，我们还会比较方案中选择的工具与候选工具，并讲述选择的理由。

接下来我们会逐个讲解上面提到的设计。

8.4　数据存储

正如前面提到的，我们的应用程序将使用 Flume 来收集原始数据，进行若干步转换操作，生成清洗后的更加丰富的数据集，以供数据分析使用。迈出这一步之前，首先需要确定原始数据、数据转换中间结果及最终数据集的存储方式。由于这几类数据集面向的需求不同，我们希望最终确定的存储选型，即数据格式及数据模型，能满足数据集的需要。

首先，由 Flume 采集到的原始数据以纯文本的格式存储至 HDFS 中。我们之所以选择 HDFS 是因为后续的数据加工需要对多种记录进行批量转换操作。正如前面章节提到的，对于扫描大数据集这样的批处理操作，HDFS 的性能不错。我们之所以选择文本格式，是因为该格式处理简单，对任何日志类型均通用，且不需在 Flume 进行额外的处理。

但是，我们对处理后的海量数据进行聚合查询和分析时，通常只涉及一部分列。分析类的查询通常需要扫描一个月的数据，最少也得扫描一天的数据。出于数据扫描性能的考虑，这里的确应该使用 HDFS。许多分析类查询每次执行时只会选择一部分列的数据，因此，对于处理后的数据，最好选择一种列式存储格式。我们使用的列存储格式是 Parquet，用来存储处理（如生成会话）后待分析的数据。考虑到 Snappy 算法对 CPU 性能有所提升，我们还会使用该算法对处理后的数据进行压缩。关于所有文件类型和压缩格式的讨论，及最优文件的选择方法，参见第 1 章。

我们计划长期存储原始数据，而不是处理后便将其删除。Hadoop 应用的设计通常都是如此。这样做有以下几点好处。

- 在 ETL 过程中，如果出现 bug 或失败，数据的重新处理会更为容易。
- 先前由原始数据生成处理后的数据集时，我们可能忽略掉了一些有意思的特征。保留原始数据，可以直接对其进行分析。另外，设计 ETL 流水线时需要浏览原始数据，决定将哪些特征包含到处理后的数据集中，所以这对于数据发现和探索是有意义的。
- 出于审计的考虑，保留原始日志也是很有意义的。

我们将采用 1.2 节中提到的目录结构来存储数据集。因为分析是基于“天”级别（一天或者多天）的，所以我们决定按照日期对点击数据进行分区，叶子节点的分区对应某一天的点击数据。对于点击数据，以下是存储原始数据和处理后数据的目录结构。

- `/etl/BI/casualcyclist/rawlogs/year=2014/month=10/day=10`

  该路径对应的是原始数据集，直接由 Flume 产生。

- `/data/bikeshop/clickstream/year=2014/month=10/day=10`

  该路径对应的是处理后的数据集，即直接供分析所用，已去除噪声的会话数据集。

如你所见，我们使用了三级（年、月、日），对以上两类数据按照日期进行了分区。

处理后的数据集大小

处理后的数据集一般比原始数据集要小一些。如果处理后的数据集以更为方便压缩的格式（如 Parquet 这样的列式存储格式）进行存储，则更是如此。于是，叶子节点分区的平均大小（如代表一天数据的分区）可能太小，不能充分利用 Hadoop。正如第 1 章中提到的那样，每个分区的平均大小应当至少是 HDFS 块大小（默认 HDFS 块大小为 64MB）的几倍，这样才能更好地利用 Hadoop 的处理能力。如果发现处理后的数据集中每个天级分区较小，那么最好不要使用天级分区，只保留年和月两级分区即可。原始数据可以仍选择年、月、日三级分区的形式。

在本章其余的内容中，我们假定处理后的数据集中，天级分区足够大，数据以三级分区的形式呈现。

另外，你可能看到过这样一种一级分区模式，形如 `dt=2014-10-10`，而不是使用三级分区模式 `year=2014/month=10/day=10`。前者实际上是另外一种有效的分区方式，这样原始数据和处理后的数据均只需拥有一个分区列（一个类型为 string 的： `dt` 列），而不是三个分区列

（列名为 year、month、day 的三个 int 类型的列）。一级分区模式使得总分区数减少，即便叶分区数等于总分区数目。分区更少，存储的逻辑元数据信息也就更少，这意味着使用访问元数据的 Hive、HCatalog 和 Impala 进行该数据集上的查询，性能可能会稍有提升。但是，一级分区模式与三级分区模式相比，前者查询时的灵活性更差。举例来说，如果在带有三级分区的数据集上进行年级别的分析的话，我们可以简单地指定 WHERE year=2014，而不是 WHERE dt LIKE '2014-%'。

不过就天级数据来看，二者均是合理的分区模式，使用起来也基本没有差别。你可以根据偏好和风格选择其中一个。本章将使用三级分区模式。

图 8-2 和图 8-3 分别展示了三级分区模式和一级分区模式。

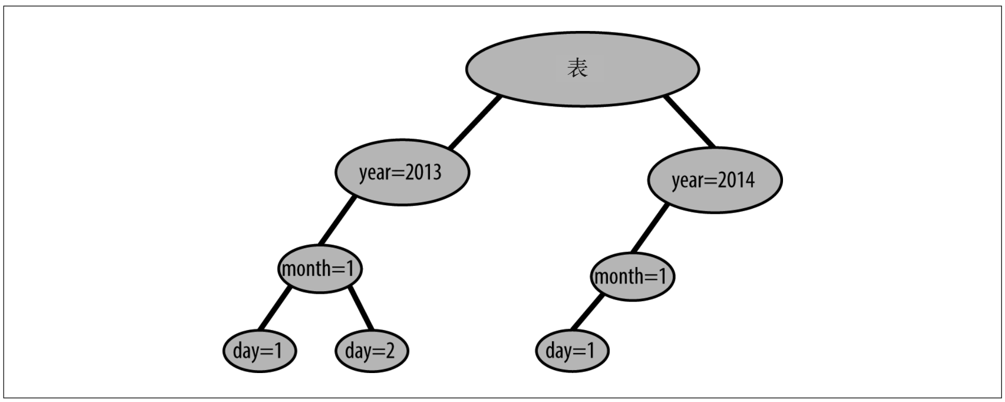


图 8-2：三级分区模式

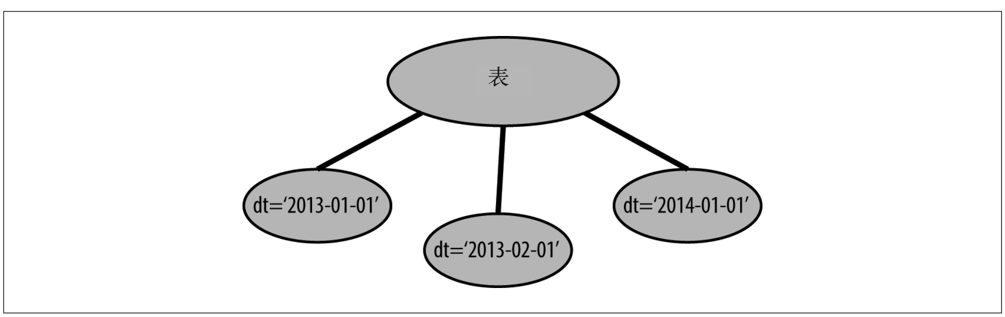


图 8-3：一级分区模式

想要了解更多关于 HDFS 和 Hive 模式设计的内容，请参考《Hive 编程指南》。

8.5　数据采集

正如我们在第 2 章中提到的那样，就导入数据到 Hadoop 而言，我们有多种方式。下面就我们的架构来评估这些工具，了解它们是否符合我们的需求。

- 文件传输

  该方式适合一次性的文件传输，对于海量点击流数据的采集并不可靠。

- Sqoop

  Sqoop 是数据导入和导出至外部存储（如关系型数据库管理系统）的利器。很明显，Sqoop 并不适合用于日志数据采集。

- Kafka

  正如第 2 章中讨论的，Kafka 的架构决定了它是将海量日志数据从网络服务器移动到各种消费者（包括 HDFS）的优秀可靠方案。因此，我们将其作为架构中数据采集选型的重要候选之一。

- Flume

  类似于 Kafka，将海量事件类型数据（如日志）移动至 HDFS 时，Flume 也是一个优秀可靠的选择。

日志采集的候选项正是 Kafka 和 Flume。二者均提供了我们所需要的可靠可扩展日志数据采集功能。我们的应用场景只需将日志数据加载到 HDFS 中，所以这里选择 Flume，因为它是面向 HDFS 的。它内置了写 HDFS 的模块，这意味着我们不需做任何定制化的开发，就可以直接搭建 Flume 流水线。Flume 还支持拦截器机制，支持数据的基本变换，比如过滤搜索爬虫的无效点击。而且，这种变换是在数据采集后、写入 HDFS 之前完成的。

如果需要建设一个更加通用的流水线，并使其支持除了 HDFS 以外的多种数据持久化方式，我们则会考虑 Kafka。

既然已经确定了数据采集的工具，下面我们讨论一下将数据导入 Flume 流水线的一些选项。

- 如果网络服务器是用 Java 语言编写的，并且使用 Log4j 记录日志，那么我们可以使用 Flume Log4j 输出将数据发送给 Flume 的 Avro 数据源。这是将事件数据发送到 Flume 的最简方式，只需将一个文件（flume-ng-sdk*.jar）加入应用环境的 Classpath 中，并对 Log4j 的配置文件稍作修改即可。不过，在本例中，我们使用的网络服务器是一个第三方的应用程序，我们无法修改其代码，或许它并不是用 Java 语言编写的，所以 Log4j 输出并不适合当前场景。
- 对于当前这种 Log4j 输出并不适合的应用场景，还是有其他一些不错的选择。你可以用 Avro 数据源或 Thrift 数据源分别给 Flume 发送 Avro 或 Thrift 消息。Flume 也可以从 JMS 消息队列中将消息拉取出来。另外一个办法是向 HTTP 数据源发送 JSON 格式的消息。选取哪种方式集成 Flume，取决于你使用的应用框架。这里有多种集成方式可供选择，便于 Flume 与已有的架构更好地融合。举例来说，如果你原本就会把事件发送到 JMS 队列中，那么就直接集成 Flume 去队列中拉取数据。如果你之前并未使用 JMS 队列，也没必要为集成 Flume 引入它，还有其他的选项可供选择。以上这些都不适合通过磁盘读取日志的应用场景。
- Flume 提供了 Syslog 数据源，这种数据源能够从 Syslog 数据流中读取事件，并将其转化成 Flume 事件。Syslog 是系统日志采集和移动的标准化工具。Apache HTTP 服务器支持将日志输出到 Syslog 中，不过仅限错误级别的日志。要将访问日志页发送到 Syslog，你需要一个变通方案，比如借助管道对访问日志重定向。

- 我们的应用场景针对一个第三方的 Web 应用，无法通过修改应用来灵活地定制化输出点击流数据，又需要处理已经存储到磁盘上的日志文件。我们来关注一下 Flume 的 Spooling Directory Source。它从一个目录中读取文件，并将文件中的每一行转换成一个 Flume 事件。注意，在网络上流传着一些使用 exec 数据源读取日志文件尾部的做法。这不是一个可靠的解决方案，很不可取。直接读取日志文件尾部，上一次读到的具体位置不会记录。Flume 程序崩溃会导致多读或者少读一些，这两种情况都是我们不希望看到的。相比之下，Spooling Directory Source 只读取已经关闭的完整文件，所以在遇到失败的情况下，它会重试整个文件并标记为成功，从而保证不会采集两次。对于从磁盘上采集数据来讲，这是一个更为可靠的方案。

我们已经选定了生成事件到 Flume 流水线的数据源，接下来需要考虑一个合适的 Flume Channel 承接上一步数据。正如第 2 章中提到的那样，如果性能比稳定性更为重要，那么 Memory Channel 就是最好的选择。如果稳定性更为重要则应当选择 File Channel。对于本章的例子，稳定性是非常重要的。在流量高峰时，Memory Channel 可能会丢失部分点击，当站点负载较高时，也可能出现这类数据丢失问题。因此，我们在 Flume 分层选型时，均选用 File Channel。

显然，我们要把数据存储到 HDFS 上，因此要采用 HDFS Sink。另外，为了更加清晰地了解处理过程，我们将数据存储为纯文本。数据会按照各自的时间戳存储在 HDFS 上的不同目录内。后面深入了解采集层的架构时，我们会继续讨论更多细节。按照日期和时间进行分区的方式能够减少数据处理时的磁盘 I/O，使数据处理任务更加高效。

现在我们已经确定了数据采集层的架构，下面来深入了解一下 Flume 流水线的架构。我们首先看一下整个顶层视图，再来认识流水线中每层的具体配置细节。图 8-4 展示了我们的采集工作流的顶层视图。在这个流程中，你会看到，我们采用的是扇入的模式（第 2 章中有所提及）。

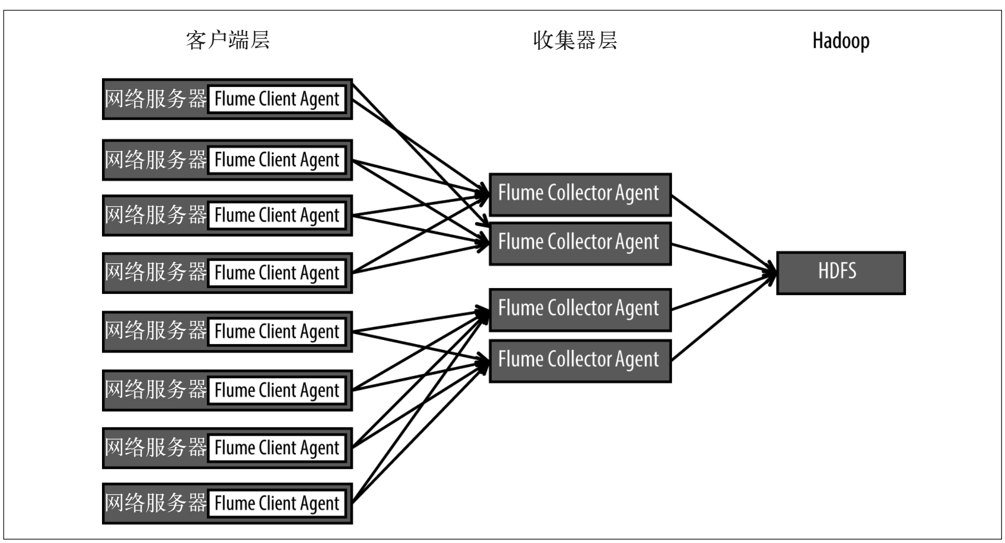


图 8-4：Flume 采集架构的整体视图

在图 8-4 中，整个流水线主要分成了三层。

- *客户端层*

  本章示例中，客户端层（client tier）由产生日志的网络服务器构成。这些日志需要采集到 HDFS 中，以便按照要求对其进行分析。此处需要注意的是，每个网络服务器的主机上都有一个 Flume agent，承担着获取网络服务器产生的日志事件，并发送到收集器层的责任。

- *收集器层*

  收集器层（collector tier）的主机能汇聚从客户端层发过来的事件，并传递到最终的目的地，即 HDFS。本章的例子假定收集器层的 agent 在集群的边缘节点上运行——这些节点在集群网络之中，它们拥有 Hadoop 的客户端配置，能够访问 Hadoop 集群并提交 Hadoop 命令、向 HDFS 写数据等。

- *HDFS 层*

  这是数据的最终目的地。收集器层的 Flume agent 承担着将事件持久化到 HDFS 文件的职责。作为持久化操作的一部分，Flume agent 要进行配置，以确保 HDFS 上文件的分区以及文件名正确无误。

在如上配置中，Flume agent 在网络服务器和收集器层上运行。注意：我们不在 HDFS 节点上运行 Flume agent，收集器层的 agent 会使用 Hadoop 客户端将事件写入 HDFS。

正如第 2 章中提到的，这种扇入式的架构设计有如下好处。

- 允许控制连接到集群的 Flume agent 数目。如果我们的网络服务器屈指可数，那么将服务器的 agent 直接连到集群中可能不会带来什么问题。但是如果有成千上百台服务器连到集群，就会引发集群的资源问题。
- 该架构能够利用收集器层节点较大的本地磁盘空间缓存事件，缓解网络服务器磁盘空间紧张的问题。
- 负载能够在多个收集器 agent 之间进行均衡。如果一个或多个收集器 agent 意外退出，该架构还可以支持事件采集的故障恢复。

接下来，我们来关注 Flume 流水线中各个层次的更多细节。

8.5.1　客户端层

图 8-5 描述客户端层 Flume agent 的具体细节。

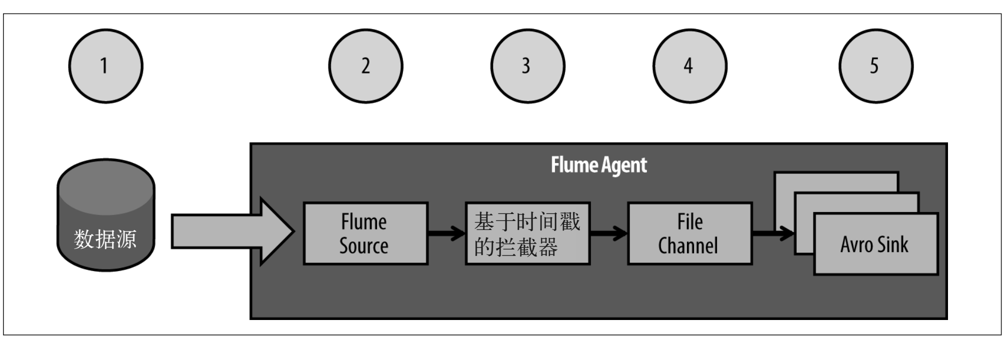


图 8-5：客户端层细节

下面介绍客户端层 Flume agent 的各个组件。

- 数据源

  数据源即指进入 Flume 流水线的事件，也就是本例中的 Apache 日志记录。正如前面提到的，本章示例中使用的是 Spooling Directory Source，所以我们会从每个网络服务器的特定目录上拉取数据源。

- Flume 数据源

  此处的 Flume 数据源即为 Spooling Directory Source，能够读取磁盘的特定目录下要采集的文件，并将其转化为事件，还会在处理完事件后重命名文件。

- 时间戳拦截器

  该拦截器的作用是向每个 Flume 事件的头部插入一个时间戳。后面的 HDFS Sink 会用到该时间戳，从而保证按照日期分区存储结果文件。

- Channel

  出于稳定性考虑，我们使用 File Channel 作为 Flume Channel。

- Avro Sink

  此处的 Avro Sink，再加上后面的 Avro 数据源，为 Flume 客户端层之间的事件传输提供了一个序列化机制。此处需要注意的是，之前我们提到的故障恢复（failover）功能就是通过设置多个 Sink 来保证的，并且由一个支持负载均衡（load balancing）的 Sink 组实现在所有可用 Sink 之间的负载分发。

> 选择 Flume 的 Avro 数据源关乎通过 HDFS Sink 存储到 HDFS 上的文件格式。正如第 1 章中提到的那样，Avro 是一种序列化格式，可用于进程间的数据传输及文件系统（如 HFDS）的数据存储。例子中的 Avro Sink 与 Avro 数据源就扮演着这样一种角色，序列化传输过程中的事件数据。以什么格式将点击数据存储到 HDFS 中，取决于最终的 Flume Sink。

下面是用于客户端层的 Flume agent 的配置文件示例。该配置文件将部署到所有的网络服务器上。

```
# 定义客户端层数据源为Spooling Directory Source：
client.sources=r1
client.sources.r1.channels=ch1
client.sources.r1.type=spooldir
client.sources.r1.spoolDir=/opt/Weblogs
# 使用时间戳拦截器向所有的事件头部添加时间戳：
client.sources.r1.interceptors.i1.type=timestamp
client.sources.r1.interceptors=i1
# 定义客户端层的Channel为File Channel：
client.channels=ch1
client.channels.ch1.type=FILE
# 定义客户端层的Sink为两个Avro Sink：
client.sinks=k1 k2
client.sinks.k1.type=avro
client.sinks.k1.hostname=collector1.hadoopapplicationarch.com
```

```
# 发送数据前配置其压缩：
client.sinks.k1.compression-type=deflate
client.sinks.k1.port=4141
client.sinks.k2.type=avro
client.sinks.k2.hostname=collector2.hadoopapplicationarch.com
client.sinks.k2.port=4141
client.sinks.k2.compression-type=deflate
client.sinks.k1.channel=ch1
client.sinks.k2.channel=ch1
# 定义一个负载均衡的Sink组,以保证在收集器层的多个节点间能够分散负载：
client.sinkgroups=g1
client.sinkgroups.g1.sinks=k1 k2
client.sinkgroups.g1.processor.type=load_balance
client.sinkgroups.g1.processor.selector=round_robin
client.sinkgroups.g1.processor.backoff=true
```

8.5.2　收集器层

图 8-6 所示为收集器层的 Flume agent 的具体细节。

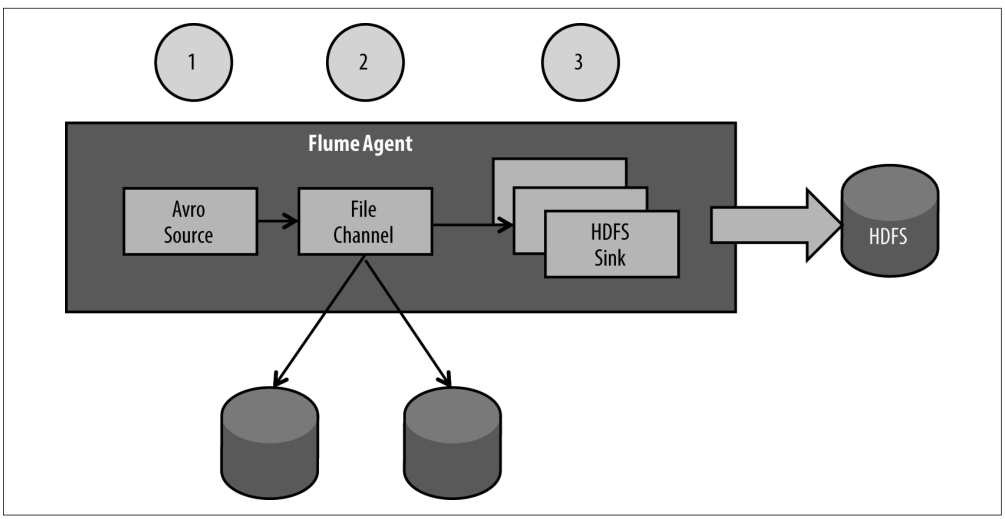


图 8-6：收集器层细节

下面介绍收集器层 Flume agent 的各个组件。

- Avro 数据源

  我们在讨论 Avro Sink 的时候就提到过，此处的 Avro 数据源就是数据在客户端层序列化后，到收集器层进行反序列化的“关键一跳”。

- Channel

  此处我们还是选择使用 File Channel 作为 Flume Channel，以确保事件在传递过程中的可靠性。为了进一步提高整体的可靠性，你可以利用集群边缘节点上的多块磁盘，详情参见第 2 章。

- HDFS Sink

  到了 Flume 流水线的最后一个环节，我们将日志事件持久化到 HDFS。值得注意的是，HDFS Sink 的配置使用 `hdfs.path`、`hdfs.filePrefix` 以及 `hdfs.fileSuffix` 等参数实现最终文件按日期分区和文件名定义。配置这些参数后，最终文件将会呈现这样的格式：/Weblog/combined/YEAR/MONTH/DAY/combined.EPOCH_TIMESTAMP.log。另外，这里要用到纯文本文件，应设置相应的 HDFS Sink 文件格式参数：`hdfs.fileType=DataStream` 及 `hdfs.writeFormat=Text`。

下面是用于收集器层的 Flume agent 的配置文件示例。该配置文件将部署到所有从属于收集器层的集群边缘节点上。

```
# 定义收集器层的数据源为Avro数据源：
collector.sources=r1
collector.sources.r1.type=avro
collector.sources.r1.bind=0.0.0.0
collector.sources.r1.port=4141
collector.sources.r1.channels=ch1
# 对收到的数据进行解压缩：
collector1.sources.r1.compression-type=deflate
# 定义收集器层的Channel为File Channel,并使用多块磁盘以提高可靠性：
collector.channels=ch1
collector.channels.ch1.type=FILE
collector.channels.ch1.checkpointDir=/opt/flume/ch1/cp1,/opt/flume/ch1/cpt2
collector.channels.ch1.dataDirs=/opt/flume/ch1/data1,/opt/flume/ch1/data2
# 定义收集器层的Sink为HDFS Sink,保证将事件以纯文本的形式写到磁盘中。
# 注意:为分散负载,配置使用了多个Sink：
collector.sinks=k1 k2
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
# 按照日期对文件进行分区：
collector.sinks.k1.hdfs.path=/Weblogs/combined/%Y/%m/%d
collector.sinks.k1.hdfs.filePrefix=combined
collector.sinks.k1.hdfs.fileSuffix=.log
collector.sinks.k1.hdfs.fileType=DataStream
collector.sinks.k1.hdfs.writeFormat=Text
collector.sinks.k1.hdfs.batchSize=10000
# 对HDFS上的文件,满足10 000条事件或达到30秒即关闭：
collector.sinks.k1.hdfs.rollCount=10000
collector.sinks.k1.hdfs.rollSize=0
collector.sinks.k1.hdfs.rollInterval=30
collector.sinks.k2.type=hdfs
collector.sinks.k2.channel=ch1
# 同样按照日期对文件进行分区：
collector.sinks.k2.hdfs.path=/Weblogs/combined/%Y/%m/%d
collector.sinks.k2.hdfs.filePrefix=combined
collector.sinks.k2.hdfs.fileSuffix=.log
collector.sinks.k2.hdfs.fileType=DataStream
collector.sinks.k2.hdfs.writeFormat=Text
collector.sinks.k2.hdfs.batchSize=10000
collector.sinks.k2.hdfs.rollCount=10000
collector.sinks.k2.hdfs.rollSize=0
collector.sinks.k2.hdfs.rollInterval=30
collector.sinkgroups=g1
```

```
collector.sinkgroups.g1.sinks=k1 k2
collector.sinkgroups.g1.processor.type=load_balance
collector.sinkgroups.g1.processor.selector=round_robin
collector.sinkgroups.g1.processor.backoff=true
```

数据采集话题的讨论即将结束，我们还要谈一下二级数据源导入 Hadoop 的问题。如果你需要关联点击数据与 CRM、ODS 或其他类似系统的数据，那么就应该把这类数据从二级数据源导入到 Hadoop 中。具体的导入方式取决于二级数据源的数据特征。本章假定这类数据存储在传统关系型数据库中。如第 2 章讨论的那样，要将数据从关系型数据库导入到 Hadoop，Sqoop 是不二之选。与点击数据相比，CRM 和 ODS 数据集的数据量很小，也不会像点击数据那样暴增（甚至不会怎么变化）。这样的数据特点决定了它们是 Sqoop 批处理任务的理想选择，即按照一天一次或一天几次的频率，将数据从 CRM 和 ODS 数据库中导入到 Hadoop。如果数据量的确不大，简单删掉后在每个 Sqoop 任务中重新导入即可。当然，如果这些数据的数据量很大，就应该使用 Sqoop 进行增量数据导入了。

8.6　数据处理

图 8-7 展示了本架构设计的数据处理部分。

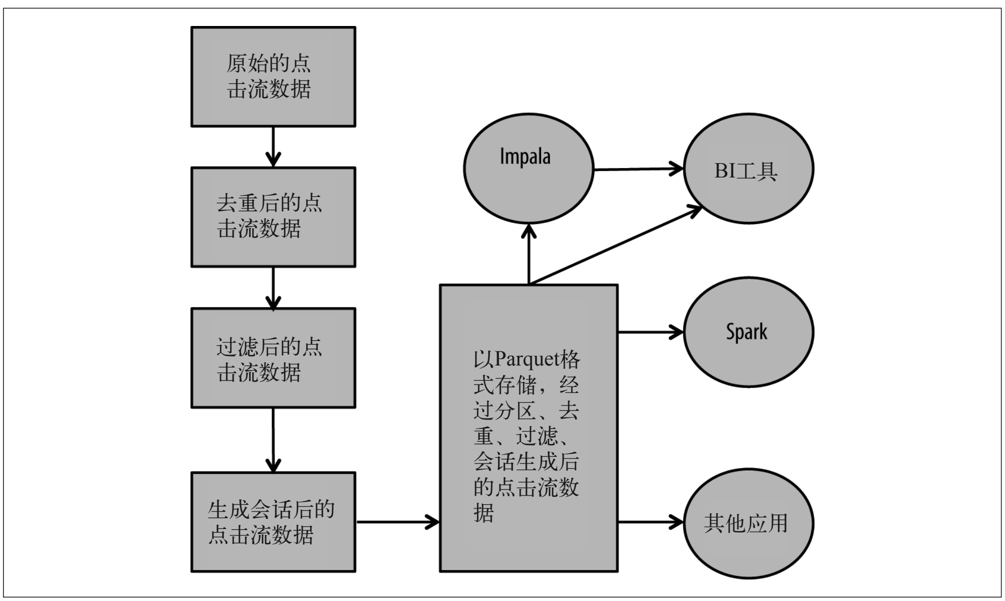


图 8-7：数据处理的设计

前面提到，点击数据通过 Flume 进入 HDFS。但是，来自网络服务器的原始数据是需要清洗的。举例来讲，不完整和无效的日志行就需要移除。此外，还可能存在一些重复的日志，需要数据去重。还有就是需要根据数据生成会话（如赋予每个点击唯一的会话 ID）。另外，有时还要对这份数据做进一步的预聚合或预分析，比如为了加速后续查询的性能，对点击进行天级别或小时级的聚合、上卷。后续查询通常会涉及市场投资回报率、贡献分

析（attribution analysis）的计算（关联营销消耗数据与数据），基于数据源的一些额外属性分析网站活跃度（需将 CRM 或其他数据源与点击关联），等等。预处理的工作需要在这一部分完成。如果进行交互式商业智能（Business Intelligence，BI）分析时还需要做数据预处理，你就会陷入手忙脚乱的窘境，这绝对不是大家希望看到的。最后，我们希望处理后的数据格式在查询过程中有不错的性能表现。

总结起来，对于数据处理的流水线，我们需要达成四点目标。

(1) *清洗*（sanitize）和检查原始数据。
(2) 从原始点击流事件中*提取*（extract）有用的数据。
(3) *转换*（tranform）提取到的数据，以生成处理后的数据集。本例需要产出的是会话数据集。
(4) 将结果集以支持高性能查询的格式*加载*（load）或存储到 Hadoop 中。

你也许已经注意到了，除了第一点的数据清洗检查，后面三点正好对应一个 ETL 流水线的提取、转换、加载步骤，这也解释了为什么要使用 Hadoop 代替 ETL。

在第一步中，我们首先需要移除不完整和无效的日志记录。简单来讲，就是使用 MapReduce 或者 Hive、Pig 这样的工具，确保每条记录（如一行日志）中每个字段均已填充，同时还可以对某些字段或者全部字段做一个快速的字段内容有效性验证。在这里，检查特定的 IP 地址或反向链接可以忽略一些垃圾点击。类似的逻辑可以添加到这个步骤，将对应的点击忽略。

接下来对日志记录进行去重。由于选用了 Flume 的 File Channel 采集数据，所以当 Flume agent 偶发崩溃时，Flume 可以保证全部的日志记录进入 Hadoop。但是，Flume 不能保证所有的日志记录均有且仅有一条，而不发生重复。一个 Flume agent 的崩溃会导致部分重复的日志记录产生，所以有必要对它们进行去重。8.6.1 节会讲解相关内容。

Hive、Pig、Spark 以及 MapReduce 均可以用来去除重复数据。其中，与 MapReduce 相比，Hive 和 Pig 的接口更高一层，实现起来更容易，同时可读性也会更好。因此，这里推荐使用 Hive 或 Pig 进行数据去重，二者皆可。究竟选取哪一个，关键在于开发者的技能背景。你需要考虑哪一个与团队的已有技术选型相符，与其他项目的特定要求相符。无关实现的具体语言，我们推荐将去重后的数据首先写入一个临时表中，然后再将其移动到最终表，以避免对原始数据集造成影响。

谈到提取步骤，我们将时间戳（自 1970 年 1 月 1 日 0 时 0 分 0 秒以来的秒数）、引用 URL 地址、用户代理字符串（包含浏览器和操作系统版本）、IP 地址、语言及 URL 等列抽取出来。之所以选择这些列，是因为我们认为基于这些数据分析结果，足够回答本章一开始提出的问题。如果后续的分析会用到所有的列，你可以不进行过滤，直接将所有的列提取出来。

在转换步骤，我们对点击进行会话生成处理，然后生成一个新的会话数据集。关于会话生成的更多内容，参见 8.6.2 节。

对于来自 ODS 或 CRM 的二级数据，一般不需对其进行数据清洗，因为这些来自关系型数据库的数据一般是正确的。将此类数据直接导入 HDFS 可以查询的位置即可。目录结构如下。

- 来自 ODS 的数据存储路径：/data/bikeshop/ods。
- 来自 CRM 的数据存储路径：/data/bikeshop/crm。

另外，一般来讲，二级数据的数据量通常较小，无需对其进行分区。

我们已经概括性地描述了整个数据处理。接下来我们看两个步骤的具体细节：数据去重与会话生成。

8.6.1 数据去重

前面提到过，如果一个 Flume agent 发生崩溃，可能产生某些重复的日志记录。

采用 Hive、Pig、MapReduce 等工具可以进行数据去重。与 MapReduce 相比，Hive 和 Pig 的接口更高一层，实现起来更为简单易读，是我们的首选。使用 Hive 的话，你需要为原始数据创建一张外表（external table），并为去重后的数据创建另外一张表，便于后续使用。使用 Pig 则不需为去重后的数据建表。重申一下，选择取决于开发者的技能背景。你要考虑哪一个与团队已有的技术选型相符，与其他项目的特定要求相符。

1. 使用Hive进行数据去重

以下 Hive 示例代码完成了点击的去重，并将产生的数据插入到按天分区的表中。

```
INSERT INTO table $deduped_log
SELECT
  ip,
  ts,
  url,
  referrer,
  user_agent,
  YEAR(FROM_UNIXTIME(unix_ts)) year,
  MONTH(FROM_UNIXTIME(unix_ts)) month
FROM (
  SELECT
    ip,
    ts,
    url,
    referrer,
    user_agent,
    UNIX_TIMESTAMP(ts,'dd/MMM/yyyy:HH:mm:ss') unix_ts
  FROM
    $raw_log
  WHERE
    year=${YEAR} AND
    month=${MONTH} AND
    day=${DAY}
  GROUP BY
    ip,
    ts,
    url,
    referrer,
    user_agent,
    UNIX_TIMESTAMP(ts,'dd/MMM/yyyy:HH:mm:ss')
  ) t;
```

此处除了抽取月份和日期信息之外，另外一个操作就是将数据集按照所有的列进行分组。如果两行记录是重复的，那么它们的所有列均相同，通过 GROUP BY 操作后就会只剩下一行。

2. 使用Pig进行数据去重

使用 Pig 去重的方法直截了当，几乎不需解释。

```
rawlogs = LOAD '$raw_log_dir';
dedupedlogs = DISTINCT rawlogs;
STORE dedupedlogs INTO '$deduped_log_dir' USING PigStorage();
```

8.6.2 会话生成

点击流分析有一个重要环节是对点击进行分组，即将单个用户单次访问（即同一个会话 ID 下）产生的多个点击聚合在一起，这种处理便称为会话生成处理。

通过分析这些生成的会话，你可以了解一次访问中，用户在网站上发生了哪些行为，包括是否购物，以及访问如何引入（如自然检索、付费搜索、友链接入等）。对于一些市场分析师来说，在一些工具中，*会话*一词的意思等同于*访问*。

你也可以通过网络服务器生成会话。在这种情况下，网络服务器会为每个点击分配一个会话 ID（代表该点击属于哪个会话）。但是，如果该会话 ID 不可靠，或者点击的生成依赖于定制化的逻辑，这里就需要自行实现（依据日志的）会话生成算法。

生成会话之前，首先需要搞清楚以下两件事情。

- 给定一系列点击，判断哪些点击来自同一个用户。只通过点击日志中的信息通常很难确定点击和用户的从属关系。一些网站会在用户首次访问其站点（或联盟网站）时，将 cookie 写入用户的浏览器。这类 cookie 也会记录到网络服务器中，因此可以据此识别来自同一个用户的点击。值得注意的是，cookie 信息不是百分之百可靠的，因为用户可能会频繁清空 cookie，这种删除甚至会在访问中途发生。还有一个方案：如果没有 cookie 信息，那么可以使用 IP 地址来识别来自同一个用户的点击。同样，IP 地址有不可靠的时候。许多公司采用网络地址转换技术（Network Address Translator，NAT），可以实现多个工作站之间的 IP 地址共享，因此多个用户可能会处理成一个用户。这时，你可以考虑利用日志中的用户代理和语言等内容进一步区分用户。这里的例子仅使用 IP 地址来识别同一个用户的点击。
- 给定特定用户的一系列点击，判断一个给定的点击是属于一个新访问的点击还是之前访问继续的点击。如果一个用户发生了三次连续点击，相邻两次之间的间隔均不超过五分钟，那么有理由认为这些点击是在一次浏览会话中发生的。然而，如果前两次点击发生在五分钟之内，而第三次点击发生在一个小时之后，那么合理的推测是第三次点击来自用户的另外一次访问，是此后的另外一个会话。大部分的市场分析工具都遵循这样的标准：如果同一用户的两次点击间隔超过 30 分钟，第二次点击应当从属于一个新的会话。另外，同许多市场分析工具一样，这里的实现也认为所有的会话会在当天结束。这样做会导致一些边界上的错误，但对问题的简化还是很有帮助的。举例来说，我们可能会重新生成某一天的会话，有了这种假设就不会影响前一天或后一天的会话了。同时，这样也会简化会话生成的处理流程。

在这一部分中，我们会举例说明如何使用不同工具实现会话的生成。Hadoop 生态圈中的 MapReduce、Hive、Pig、Spark、Impala 等众多工具均可以实现该处理，第 3 章介绍过它们。通常情况下，你可以选择其中的一种来生成会话数据集。Hadoop 中的绝大多数框架都可以满足需求，因此工具选择的主要考量仍在于开发者的技能背景，看哪一个与团队已有的技术选型相符，哪一个能满足其他项目及性能需求。

总之，任何会话生成算法都包含以下步骤。

(1) 遍历整个数据集，提取每个点击对应的相关列（本例中指 IP 地址，也可加上 cookie）。
(2) 将单个用户当天的所有事件收集起来，形成按照时间戳分类的用户维度的序列。
(3) 遍历每个用户序列，对序列中的每一个点击事件分配一个会话 ID。如果连续点击行为之间的间隔超过了 30 分钟，则会话 ID 自增。

接下来，我们来了解如何使用不同的工具实现这些步骤。

1. 使用Spark生成会话

Spark 是 Hadoop 上的一个新执行引擎，其总体性能比 MapReduce 要好。我们使用 HDFS 上的点击数据创建一个 RDD。然后通过 `map()` 函数提取每个点击的重要字段，以键值对的方式返回，以 IP 地址为键，以抽取生成的日志字段对象作为值（第一步）。以下为该函数的示例代码。

```
JavaRDD<String> dataSet =
  (args.length == 2) ? jsc.textFile(args[1]) : jsc.parallelize(testLines);

JavaPairRDD<String, SerializableLogLine> parsed =
  dataSet.map(new PairFunction<String, String, SerializableLogLine>() {
    @Override
    public Tuple2<String, SerializableLogLine> call(String s)
      throws Exception {
        return new Tuple2<String, SerializableLogLine>(getIP(s), getFields(s));
        }
});
```

接下来，我们将从属于同一个 IP 地址的点击分为一组，并按照时间戳进行排序（第二步），然后使用 `sessionize()` 方法遍历点击的有序列表，并给每一个点击分配会话 ID（第三步）。以下为上述步骤的代码。

```
//Sessionize生成函数,输入一个IP的事件序列,
//按照时间戳进行排序,并将30分钟以内的所有事件标记为同一个会话
public static List<SerializableLogLine> sessionize
        (List<SerializableLogLine> lines) {
    List<SerializableLogLine> sessionizedLines = Lists.newArrayList(lines);
    Collections.sort(sessionizedLines);
    int sessionId = 0;
    sessionizedLines.get(0).setSessionid(sessionId);
    for (int i = 1; i < sessionizedLines.size(); i++) {
        SerializableLogLine thisLine = sessionizedLines.get(i);
        SerializableLogLine prevLine = sessionizedLines.get(i - 1);

        if (thisLine.getTimestamp() - prevLine.getTimestamp() > 30 * 60 * 1000) {
            sessionId++;
```

```
            }
            thisLine.setSessionid(sessionId);
        }

        return sessionizedLines;
        }

    // 此处按照IP地址对点击进行分组
    JavaPairRDD<String, List<SerializableLogLine>> grouped = parsed.groupByKey();

    JavaPairRDD<String, List<SerializableLogLine>> sessionized =
            grouped.mapValues(new Function<List<SerializableLogLine>,
                    List<SerializableLogLine>>() {
    @Override
    public Iterable<SerializableLogLine> call
            (List<SerializableLogLine> logLines) throws
            Exception {
       return sessionize(logLines);
        }
    });
```

你可以访问本书的 GitHub 仓库，获取使用 Spark 实现会话生成的完整代码。

2. 使用MapReduce生成会话

使用 MapReduce 生成会话，可以从代码层次获得更深一层的控制。使用 `map()` 函数可以获取点击日志中的相关字段（第一步，参见前面提到的三个步骤）。使用 `shuffle()` 函数可以按照用户对事件进行集合和分组，并使用一个定制化的比较器（comparator）实现点击列表按照时间戳排序的需求，然后将数据发送给 Reducer 端（第二步）。`reduce()` 函数会遍历每个用户有序点击的列表，并分配会话 ID（第三步）。

你可以访问本书的 GitHub 仓库，获取使用 MapReduce 实现会话生成的完整代码。

3. 使用Pig生成会话

Pig 中的常用库 DataFu 包含一个 `Sessionize` 函数。该函数以特定用户特定日期的记录列表作为输入，每条记录的第一个字段是时间戳，点击列表以时间戳升序排列。`Sessionize()` 函数的输出是一个点击记录列表，每条记录都包含会话 ID。

4. 使用Hive生成会话

虽然仅使用 SQL，利用 SQL 的窗口分析函数的确可以实现会话的生成，但这样难以维护和调试写出的查询。我们不推荐使用 Hive 或 Impala 处理该任务。

8.7　数据分析

在采集和处理完数据之后，就可以进行数据分析，回答本章开头提到的问题了。业务分析人员可以使用若干工具来浏览和分析数据。第 3 章对此有详细介绍，简而言之，这些工具可以分成如下三类。

- 可视化及 BI 工具，如 Tableau 和 MicroStrategy。
- 统计分析工具，如 R 或 Python。
- 面向机器学习的高级分析工具，如 Mahout 或 Spark MLlib。

- SQL 接口工具，如 Impala。

本章主要关注如何通过 Impala 对处理后的数据集进行 SQL 查询访问和分析。

举例来说，想知道购买者在该网站上平均花了多长时间，可以执行以下查询。

```
SELECT
  AVG(session_length)/60 avg_min
FROM (
  SELECT
    MAX(ts) - MIN(ts) session_length_in_sec
  FROM
    apache_log_parquet
  GROUP BY
    session_id
  ) t
```

使用如下查询，还可以计算网站的跳出率（用户打开本网站后尚未跳转至其他页面便已结束访问的百分比）。

```
SELECT
  (SUM(CASE WHEN count!=1 THEN 0 ELSE 1 END))*100/(COUNT(*)) bounce_rate
FROM (
  SELECT
    session_id,
    COUNT(*) count
  FROM
    apache_log_parquet
  GROUP BY
    session_id)t;
```

我们还可以把 BI 工具（如 MicroStrategy 或 Tableau）连到 Impala 上，通过 Impala 提供的 ODBC 或 JDBC 驱动，针对点击数据进行更进一步的 BI 查询。

8.8 协调调度

到这里，我们已经将数据采集到了 Hadoop，并针对点击流数据进行了各种各样的处理，最后以终端用户的身份分析处理后的数据。最终的数据分析是以即席查询的方式进行的，但是前面的部分（数据的采集和各种处理行为）应当能够通过多步骤协调、调度的方式自动完成。在这一部分，我们将展示如何协调调度基于 Hadoop 进行点击流分析的各种步骤。

在数据采集方面，我们使用 Flume 获取源源不断的数据，并将其导入系统。在数据处理方面，我们则决定每天运行一次会话生成算法，因为在一天结束的时候会话一般也结束了。考虑到数据延迟与算法复杂性之间的权衡，可以每天运行一次会话生成程序。如果想更频繁地生成会话，则需要维护一个正在运行的会话的列表（因为会话可以持续任意时长），会话生成算法会变得过于复杂。这对于近实时系统来讲是非常慢的，但并不是所有的系统都需要做到实时。第 9 章详细介绍了如何搭建一个近实时分析系统，而且第 9 章中的许多技术也都适用于点击流分析的应用场景。

我们使用第 6 章中提到的 Oozie 协调与调度本章的会话生成处理。

在本章示例中，Oozie 工作流会先做一些预处理工作，然后再生成会话。以下是 Oozie 工作流的示例，包括使用 Pig 处理数据去重，以及使用 MapReduce 生成会话。

```
<workflow-app xmlns="uri:oozie:workflow:0.4" name="process-clickstream-data-wf">
  <global>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
  </global>
  <start to="dedup"/>
  <action name="dedup">
    <pig>
      <prepare>
        <delete path="${dedupPath}"/>
      </prepare>
      <script>dedup.pig</script>
      <argument>-param</argument>
      <argument>raw_log_dir='${wfInput}'</argument>
      <argument>-param</argument>
      <argument>deduped_log_dir='${dedupPath}'</argument>
    </pig>
    <ok to="sessionize"/>
    <error to="fail"/>
  </action>

  <action name="sessionize">
  <java>
    <prepare>
      <delete path="${sessionPath}"/>
    </prepare>
    <main-class>com.hadooparchitecturebook.MRSessionize</main-class>
    <arg>${dedupPath}</arg>
    <arg>${sessionPath}</arg>
  </java>
  <ok to="end"/>
  <error to="fail"/>
  </action>

  <kill name="fail">
    <message>Workflow failed:[${wf:errorMessage(wf:lastErrorNode())}]</message>
  </kill>
  <end name="end"/>

</workflow-app>
```

有一点需要注意，我们通过传递 `year`、`month`、`day` 参数形成 `dedupPath` 变量的值，这几个参数确定了要生成的点击数据的确切日期。它们的值通过负责调度该工作流的协调器任务产生。

在我们的设计中，协调器每天都会触发工作流，触发后该工作流会处理前一天的数据。我们需要确保前一天的数据在进行会话生成时已经收集完毕。这样保证了数据处理前的同步性，是非常常见的流程协调模式。如果在处理当天数据时，这些数据很还没有写完，那么就会出现不一致的情况。前面提到的模式可以避免这一点。

有两种方式可以保证上文提到的同步。

- 在开始处理前一天的工作流之前，验证 Flume 已经开始写当天的数据。
- 等数据集写完，生成一个文件标识，让 Oozie 等到该标识生成完毕再继续执行处理。文件标识通常名为 _SUCCESS，表明整个数据集已经成功落地，可以开始后续处理。协调器支持这样的调度规则，即等到指定目录下文件标识生成才执行工作流（使用 `<done-flag>` 选项）。

在本例中，Flume 不太容易生成一个文件标识。相对而言，检查当天的数据是否已经开始写入是容易实现的，只需检查当天的分区是否存在即可。因此，上文提到的第一种方式就可以保证同步了，协调器这边对应的代码示例如下（下文代码有部分截断，全部代码参见本书 GitHub 仓库）。

```
<coordinator-app name="prepare-clickstream" frequency="${coord:days(1)}"
                 start="${jobStart}" end="${jobEnd}"
                 timezone="UTC"
                 xmlns="uri:oozie:coordinator:0.1">

  <datasets>
    <dataset name="rawlogs" frequency="${coord:days(1)}"
             initial-instance="${initialDataset}" timezone="America/Los_Angeles">
      <uri-template>/etl/BI/casualcyclist/clicks/rawlogs/year=${YEAR}/...
      <done-flag></done-flag>
    </dataset>
  </datasets>

  <input-events>
    <data-in name="input" dataset="rawlogs">
      <instance>${coord:current(0)}</instance>
    </data-in>
    <data-in name="readyIndicator" dataset="rawlogs">
      <!-- Flume完成某一目录的写操作之后并不会设置写完标识,
           无法得知该目录何时可以作为输入,因此直到开始写第二天的数据,
           才开始通过协调器来调度工作流 -->
      <instance>${coord:current(1)}</instance>
    </data-in>
  </input-events>

  <action>
    <workflow>
      <app-path>${workflowRoot}/processing.xml</app-path>
      <configuration>
        <property>
          <name>wfInput</name>
          <value>${coord:dataIn('input')}</value>
        </property>
        <property>
          <name>wfYear</name>
          <value>${coord:formatTime(coord:dateOffset(
          coord:nominalTime(), tzOffset, 'HOUR'), 'yyyy')}</value>
        </property>
        <property>
          <name>wfMonth</name>
```

```
          <value>${coord:formatTime(coord:dateOffset(
          coord:nominalTime(), tzOffset, 'HOUR'), 'MM')}</value>
        </property>
        <property>
          <name>wfDay</name>
          <value>${coord:formatTime(coord:dateOffset(
          coord:nominalTime(), tzOffset, 'HOUR'), 'dd')}</value>
        </property>
      </configuration>
    </workflow>
  </action>
</coordinator-app>
```

如你所见，`readyIndicator` 告诉 Oozie，开始写第二天的数据时才开始执行工作流，这意味着前一天的数据已经采集完毕。

8.9 小结

在本章讲解了一个常见的 Hadoop 使用情境：以批处理方式分析机器产生的数据。存储和处理如此海量、高吞吐、多样性的数据，使用 Hadoop 再合适不过了。

我们的架构设计使用 Flume 采集点击流数据，使用 Sqoop 将来自 CRM 或 ODS 的二级数据源导入数据集。接下来，我们讨论了基于点击数据的常见处理，涉及数据去重、数据过滤以及最重要的会话生成。另外，这里还描述了生态系统中各种不同的执行引擎是如何满足这些处理需求的。后面，我们展示了基于 Hadoop 上的数据集进行的分析类查询，如查找一个网站的跳出率。最后，我们演示了如何通过协调调度工具（如 Oozie）组织整个工作流。

虽然本章提到的架构是面向批处理的，但面向近实时处理的点击流分析也可以通过将数据存储到 NoSQL 系统（如 HBase）中实现。本章未讨论该方案，不过本书的 GitHub 仓库中有架构设计和代码可供参考（http://bit.ly/haa-session）。

或许本章提到的案例跟你遇到的实际情况并不完全相符，不过基于 Hadoop 进行机器数据处理大抵皆是如此。我们希望本章能够在设计 Hadoop 应用方面穿针引线，给你提供一些帮助。

第 9 章
欺诈检测

什么是欺诈检测？这是一个价值几十亿美元的产业，关乎任何一家公司的大额资金损失。欺诈检测的核心究竟是什么？出于讨论所需，在这里我们可以认为欺诈检测就是基于行动者（人或机器）是否按照其应有的行为行动而进行决策。这里包括两类问题：其一，知晓正常行为与异常行为的不同，其二，根据这些知识有所行动。

关于第一点我们举一个简单的例子：家长总是能知道孩子是否在撒谎或有所隐瞒。这一点，有孩子的人都有体会：家长要每天照顾孩童，所以知晓孩子正常的时候是什么样子。如果有一天，孩子忽然变得比平常安静，或者总是不敢看大人的眼睛，那么家长凭直觉就能发现问题。父母会提出各种问题，直到发现孩子考试失败、跟朋友打架或者在学校受了欺负之类的事情。家长之所以能够发现这样的隐瞒跟欺骗，是因为他们与孩子之间有紧密的亲子关系，因为家长了解孩子。行为模式成熟后，这种亲近的关系是监测变化的关键。

9.1　持续改善

现在，事情很少会闹到孩子盗窃银行上百万或卷入洗钱案件的地步。不过同编程中的情况一样，孩子总会试图掩饰些什么。随着年龄的增长，孩子的说谎能力也提高了。与此相同，骗子与黑客也越来越精通欺诈。

人们会以不同的水平处理新信息并作出决策。在这里，我们简单分出两个不同的处理组：背景处理与快速反应。背景处理用于学习与提高处理，而基于现存的处理作出反应时，我们使用快速反应。

回到孩子的例子中，了解一个甚至几个孩子与了解数十亿的信用卡持有者、顾客、银行账户所有者、电子游戏玩家或者粉丝等从规模上来讲完全不同。Hadoop 能在这里提供有价

值的服务，以足够的空间存储所有需要的人物相关信息，而且 Hadoop 能够检阅所有人物透露出的信息。

9.2　开始行动

有了背景处理作为基础，当事情看起来有点奇怪时，我们已经准备好作出反应。在这种场景中，快速反应就是观察到你的孩子比平常更安静或者躲避眼神交流，你从直觉上感觉到有些事情不太正常。但是，这种快速反应处理只能发生在你了解孩子的正常行为后。作为父母的我们要将孩子的行为与档案比对，需要检测异常时迅速作出反应。

有意思的是，了解大脑学习与反应的方式后，我们能够创建一个成功的欺诈检测系统。欺诈检测系统应该能够迅速作出反应，并且基于到来的数据进行较小程度的更新，但是需要线下处理发现观点。这些观点可能会改变快速反应阶段的规则，或者发现新的反应模式。

上面已经讨论过信用卡交易欺诈，但是应该注意，欺诈检测涉及各个行业与领域。与我们之前注意到的相同，任意一种依靠欺骗获利的行业都是有风险的。在后面的研究案例中，我们会使用银行或者信用卡公司使用的欺诈检测系统来检测针对消费者账户的欺诈。类似的架构也适用于欺诈检测应用，比如在一个大型多人网络游戏中检测外汇交易或商品交易欺诈。

9.3　欺诈检测系统架构需求

欺诈检测系统与其他很多 Hadoop 使用案例有所不同，前者包括快速反应组件。点击流分析系统或者数据仓库可能需要处理数十亿的事件，但是可以花费数小时来做。欺诈检测系统必须在数毫秒之内对事件作出反应，并且必须完全可靠。除此之外，这里还需要每秒处理数百万交易，并且在背景中运行大规模的数据分析。Hadoop 生态系统提供实时组件（如 Flume、Kafka 与 HBase）、近实时组件（如 Spark Streaming）以及搜索组件，能够用于背景处理与分析（如 Spark、Hive 与 Impala）。

9.4　用例介绍

为了帮助你理解欺诈检测架构，我们创建了一个简单的例子，该例子实现了在银行账户或者信用卡方面的欺诈检测。

本例中，欺诈检测的目的是检阅进入的新交易。如果这个交易比用户每日正常的交易数额大很多，它就会被标记。系统也会检测用户的位置是否与过去 20 个位置有所不同。在网络交易系统中，这意味着检查 IP 地址是否与过去 20 个登录地址有所不同。在实际销售点系统，这意味着检查用户的地理位置（如城市）是否与过去 20 个位置有所不同，而且按照就近原则。如果交易发生在一个新的地址（线上或其他），那么交易平均值的资金限额会被进一步降低。

这是一个简单的例子，使用一张画像表来存储账户持有者的信息，实现本地缓存层，通过 HBase 进行后端持久性处理。尽管这个例子很简单，但它能让你理解如何使用 Hadoop 进

行欺诈检测，也提供了一个在 Hadoop 上进行近实时处理的具体案例。

> **异常检测技术的广泛关联性**
>
> 注意，我们将欺诈检测定义为“基于行动者（人或机器）表现是否正常作出决策”。区分正常与异常事件的能力也被称为异常检测（anomaly detection 或 outlier detection）。异常检测不仅能够防止金融欺诈，还在很多行业有着深远的影响。本章所示的技术能够用于检测安全系统遭受的入侵、机器错误的提前警告、为了再次及时获取而可能放弃服务的消费者，或者处方药的非法购买（药物滥用的提前预警）。无论处于哪个行业，异常检测都是最有用的数据分析之一。

9.5 架构设计

那么，Hadoop 是如何帮助我们实现这样一个系统的呢？这种系统有多种创建方法，我们首先看一个一般设计，而后深入探讨不同的组件选择，以便解释整个设计如何根据特定的使用案例进行调整。

图 9-1 为我们的系统提供了一个整体的架构图。这张图较为复杂，我们将其分开，分部分讲解。

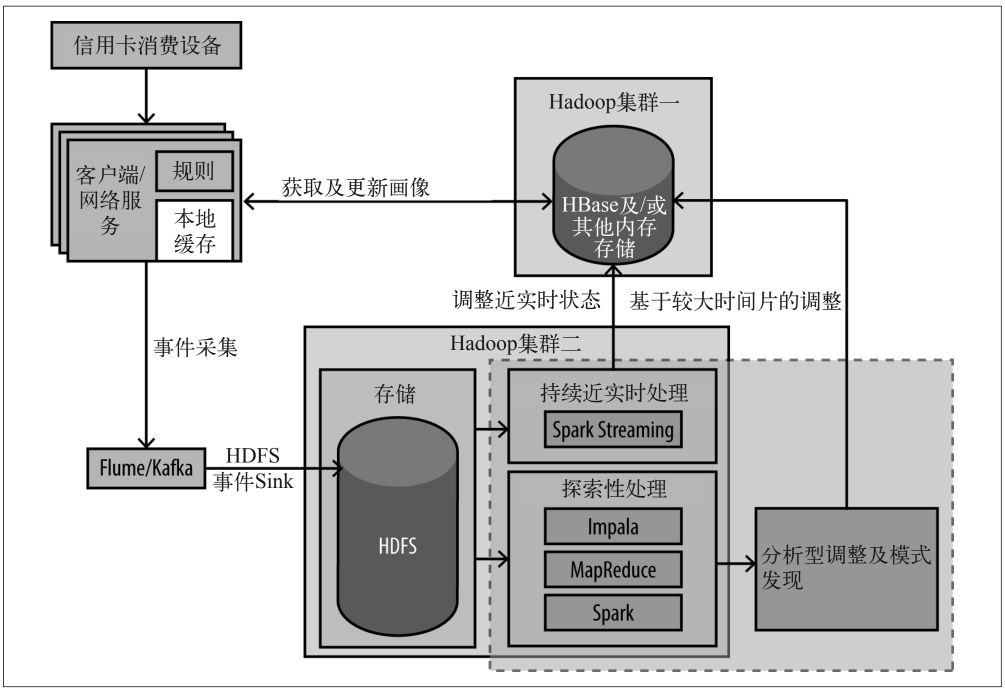


图 9-1：欺诈检测系统的架构图

- 我们首先介绍左上角的客户端（client）。在这个例子中，客户端是一种网络服务，负责接收信用卡交易请求并作出反应（同意或拒绝）。这里有多个格子，因为 Web 服务可能分散在多个服务器上。
- 为了同意一项交易，Web 服务可能需要从 HBase 中检索用户画像与交易历史，也需要在 HBase 中记录目前的交易，将交易作为用户历史的一部分。
- 交易、同意或者拒绝的决定以及决定的原因都会通过 Flume 发送到 HDFS 汇总。
- 存储于 HDFS 时,数据能够使用近实时的处理框架如 Spark 或 Spark Streaming 自动处理。Spark Streaming 也能直接从 Flume 中处理数据。NRT 处理能够监测趋势以及过去几分钟之内的事件，这些无法直接通过单独观察每个交易（例如，在同一个地理位置或者同一家商店中多个用户的可疑活动）而看出端倪。NRT 处理能够在 HBase 中相应地自动更新画像。
- 人们也能够使用工具探究与分析数据，如 Impala、Spark 与 MapReduce。这些工具能够发现其他行为模式并同时在 HBase 中更新客户端逻辑与信息。

本章将详细讲解客户端架构、HBase 模式设计以及基于 Flume 的采集架构。我们将给出一些近实时处理的建议与探索性分析，但是特定机器学习算法不在本书范围之内。

机器学习资源推荐

机器学习算法知识对于任何一个认真看待欺诈检测系统实现的人来说都非常实用。以下资源可以帮助你展开这方面的学习。

- 如果想在数据分析方面找到有趣又涨知识的资料，我们推荐你关注 Analytics Made Skeezy（http://analyticsmadeskeezy.com/）。这个博客采用一种比较风趣的方式，讲述了与欺诈犯罪有关的主要方法和算法。
- 如果你想在机器学习方面找到比较实用的方法，我们推荐 Douglas G. McIlwraith 等人所著的 *Algorithms of the Intelligent Web*。这本书讲述的算法使用了易于理解的 Java 实现，让一些流行网站更加强大。
- 如果你想获得实用介绍，并且倾向于使用 Python 而不是 Java，那么我们建议你阅读 Sarah Guido 即将出版的书 *Introduction to Machine Learning with Python*。
- 如欲了解更多深层次的理论背景，又不打算在数学专业修一个博士学位，那么我们建议你阅读 Stuart Russell 与 Peter Norvig 合著的 *Artificial Intelligence: A Modern Approach*。

下面简单介绍一下后面章节的主题。

- 客户端架构：
  - 客户端如何对到来的事件作出决策？
  - 客户端如何使用 HBase 存储、检索与更新用户画像信息？
  - 客户端如何将交易状态传送给其他系统？
- 采集：如何使用 Flume 将客户端的交易数据采集到 HDFS 中？
- 近实时处理：如何学习低延迟处理的到来数据？
- 背景处理、检阅以及调整：如何学习所有的数据并升级处理？

你也可以使用本章结束时详细讨论的其他方法，但本章重点介绍的这些更为易于使用，具有良好的可伸缩性，而且使用时非常快速。

现在详细看一下构架中的子系统。

9.6 客户端架构

在系统中，交易的欺诈检测逻辑在客户端实现。客户端自身可以是一个 Web 服务器或者类似的应用，即任意一种需要检阅事件并作出决策的系统。在这里的讨论中，我们假设这是一个 Web 服务器，它的一个主要任务是检阅到来的事件并反馈同意或拒绝。

客户端如何精确地执行所有的查询并报警？可以分三次检查每个事件。

- 事件验证

  这是指在事件本身的上下文中进行规则验证，比如格式或基本逻辑规则的验证（“列 A 必须是一个正数”“列 B 表示撤回”，诸如此类）。

- 全局上下文验证

  这是指在全局信息上下文中的规则验证，如风险阈值验证。此处可能需要验证 URL 是否安全。你可能拥有一个 IP 地址的全局列表，并含有风险分级，而规则可能只允许 IP 低于一个特定风险值。

- 画像内容验证

  这种验证提高所需要的事件相关的行动者（actor）的信息水平。银行交易就是一个很好的例子。如果用户最近 100 个交易都发生在俄亥俄州与密歇根州，俄亥俄州一次交易后 20 分钟，我们突然收到一个发生在俄罗斯的交易，那么我们就有理由报警。

综合所有这些验证，客户端应用每收到一个事件都会检索画像，使用画像与事件执行欺诈检测逻辑并返还结果。

注意，正因为需要全局级别和画像级别的验证，这里会用到 Hadoop 或 HBase。如果只是需要事件验证，那么完全可以将 Hadoop 或 HBase 从架构中移除。这里涉及数十亿潜在用户的画像信息并且要不断更新，所以 Hadoop 与 HBase 非常适合。

让我们把重点放在需要时如何将整体的用户画像信息填充到客户端。

9.7 画像存储及访问

在存储用户画像以及从客户端访问画像方面，有两大挑战。第一，需要存储的数据量很大，可能涉及十亿用户的长期交易历史。第二，信息需要快速获取。为了及时地响应欺诈，我们需要保证在毫秒级内返回任意用户的历史查询。本节会研究使用 NoSQL 时的一些选择，并为信息的快速获取配置缓存。我们会关注 HBase，讲述如何将用户画像存储到 HBase 中，并保证快速地获取和更新数据。

9.7.1 缓存

客户端获取画像信息的最快方式是从本地内存中获得。如果本地缓存里有画像，那么就可以以亚微秒级别的延迟获取验证逻辑需要的信息。

讨论如何完整地实现一个良好的缓存层显然超出了本书的范围，不过我们会对实现缓存框架背后的基本原理进行整体性地快速介绍。

任何缓存框架的核心都是内存。内存当然有其自身的限制。指定节点上很可能没有足够的内存装下我们所需的全部数据。我们推荐使用多层方式，将本地缓存与远程的分布式缓存结合在一起。

不管是否会用到额外的缓存层，我们都推荐为大多数活跃的用户使用本地缓存。虽然这样做会增加系统扩展的复杂性，但本地内存的延迟（亚微秒级）与经由网络读取画像的延迟（至少为毫秒级）不同，这种复杂性的增加是值得的。为了克服本地缓存的内存大小限制，我们可以对客户端之间的画像信息进行分区，从而保证每一个客户端保存记录的一个子集，业务判定的请求发送到合适的客户端。我们的例子使用 Google 的 GUava 缓存库，该库能够方便地创建本地缓存（https://github.com/google/guava/wiki/CachesExplained），并可以配置它的 `get` 方法以保证缓存中没有画像时从 HBase 中加载。

有以下两种关于远程缓存层的可能。

- 以分布式内存缓存方案（如 Memcached）在集群节点中分发数据。
- 配置 HBase，以保证能够在块缓存中找到所有需要的记录。块缓存能够将数据块保存在内存中，进行快速访问。

下面我们将继续讨论这两点。

1. 分布式内存缓存

类似于 Memcached 或 Redis 的分布式内存解决方案简化了缓存层的开发工作。就性能而言，这里仍需要进行网络调用，会略微拉长请求的延迟。请求时间大概为 1~4 毫秒。这一方案与分区方案相比，优点在于节点发生故障时不会有停机时间，这是因为我们会配置好缓存系统中数据的多个副本。这一方案的缺点在于，需要有足够大的内存来装下所有的东西。如果无法承载所有的数据，那么就需要后端挂一个磁盘作为额外的持久化存储。这也就意味着当数据不在内存中时，需要额外调用。稍后我们会看到，使用了 HBase，便几乎就没有理由使用分布式的缓存方案了。

2. HBase的BlockCache

配置无误的话，HBase 作为缓存层可以提供毫秒级的响应时间，只要我们将要查找的结果保持在内存中。在这种情况下，我们会使用 HBase 的 BlockCache，保证最近使用的数据块保存在内存中。注意，HBase 先前的版本在内存上有限制，会影响 BlockCache 的大小，不过最近的改进已经消除了许多这样的限制。这些改进包括对 BlockCache 的 OffHeap 处理、对 BlockCache 进行压缩以及对 Java 的垃圾回收（Garbage Collection，GC）系统进行优化。

然而，HBase 的一个核心特点（强一致性）对欺诈检测系统的近实时响应有着潜在的不利影响。就像 CAP 理论中提到的那样，类似于 HBase 这样保证强一致性和分区容忍性的系

统提供不了多少可用性保证。实际上，如果一个 HBase 的 Region 服务器发生故障，在一段时间内（可能持续几分钟），会有一些行键无法读取或者写入。

出于这样的原因，使用 HBase 作为核心组件的生产环境需要引入一种机制来加强可用性。这个话题超出了本书的范围，不过这方面的工作——在 HBase 项目上支持多个 HBase 集群运行并保证集群之间的数据同步——还在进行中。这样做能够在持久化失败或从主 HBase 集群获取数据时做到故障切换。

9.7.2 HBase数据定义

为了进一步了解怎样使用 HBase 实现一个解决方案，现在来看一下数据模型以及与 HBase 的交互。在信用卡交易欺诈检测的例子中，我们需要一个含有信息足够多的画像，这样在运行模型的时候才能判断是否存在行为异常。

以下 Java 代码代表 `ProfilePojo` 类，我们会在后面讨论它的值。注意，我们将这些字段分成这样几个类别：几乎不会发生更改的字段、经常发生更改的字段、求和的字段、捕获历史信息的字段。

```
// 几乎不会发生更改的字段：
  private String username;
  private int age;
  private long firstLogIn;

// 频繁更改的字段：
  private long lastLogIn;
  private String lastLogInIpAddress;

// 计数器：
  private long logInCount;
  private AtomicLong totalSells;
  private AtomicLong totalValueOfPastSells;
  private AtomicLong currentLogInSellsValue;
  private AtomicLong totalPurchases;
  private AtomicLong totalValueOfPastPurchases;
  private AtomicLong currentLogInPurchasesValue;

// 记录历史信息的字段：
  private Set<String> last20LogOnIpAddresses;
```

在 HBase 上，不同类型的存储情况可能不同，因此需要进行这样的归类。现在看一下有哪些方式可以实现 HBase 数据字段的持久化和更新。

1. 列（组合列或原子列）

对于前两类，我们关注的是对应值不怎么变化和经常发生变化的情况。我们可以将这些列存储到同一列或者各存一列。

使用 RDBMS 的用户从来不会考虑将多个字段存储到一列中，但是 HBase 有所不同，它会以另一种方式将数据存储到磁盘上。图 9-2 展示了 HBase 将五个值存储在一列的情况，以及将它们分别存储在不同列的情况。

行键	时间戳	列	值
42	123456789	c	JimS|80|123456789|123456788|1.0.0.127

行键	时间戳	列	值
42	123456789	un	JimS
42	123456789	ag	80
42	123456789	fLg	123456789
42	123456789	|Ln	123456788
42	123456789	|Lg|p	1.0.0.127

图 9-2：同一列存储与单独列存储

如图 9-2 所示，使用同一个列的好处在于节省磁盘空间。现在，在单独的列中，大多数的额外信息会被压缩，不过仍然会有解压缩和经由网络发送的损耗。

将所有的值组合在一起，这样做的缺点在于无法原子性地更改它们。因此，组合列适用于那些几乎不变或者通常总是一同更改的列。这就是说，由于在这一方面存在着性能的优化点，最好能够在开发伊始将事情简化。一个需要遵从的设计原则就是封装进出 HBase 的持久化和序列化对象，从而保证模式上的任何更改仅在自己的代码中产生影响。

使用短列名

需要指出的是，图 9-2 涉及短列名的使用。你应该记得，列名也会存储到磁盘上，但 RDBMS 往往未必如此。记住，HBase 没有固定的模式，每条记录可以有不同的几列。于是，每条记录还需要对应列的名称。理解了这一点，就把列名缩短吧。

2. 使用HBase的`increment`或`put`方法作为事件计数器

HBase 的 API 提供了一个 `increment` 方法，可以允许用户对 HBase 上存储的一个数值型的值进行增加操作。这个值是根据行键、列簇名、列名确定的。举例来说，如果这个值之前是 42，对其进行加 2 操作，那么就会变成 44。

这一功能的强大之处在于，多个应用可以对同一个值进行增加操作，确保以线程安全的方式增加该值。熟悉 Java 的人应该知道，这里的增加操作类似于 `AtomicLong` 和 `addAndGet()` 方法，但是这与简单的 `long++` 不同，因为后者不是线程安全的。

不过，尽管 `increment` 是线程安全的，仍然有一个需要考虑的问题——不是 HBase 的问题，而是客户端的问题。很容易想到的一点是，客户端发送了 `increment` 请求，也许未得到先前事件的确认就发生故障了。在这种情况下，客户端会尝试恢复并再次执行 `increment`。结果 `increment` 便执行了两次，也就是说计数在重置之前是错误的。这一问题在更新多个 HBase 集群时也会出现：向主集群发送 `increment` 请求，该请求花费了太长时间，客户端以为该请求失败，然后将 `increment` 发送给故障切换的集群。如果第一个 `increment` 执行

成功，那么 `increment` 就多执行了一次。

考虑到以上潜在问题，采用 `increment` 可能不是最好的方式。另外一种方式采用 HBase 的 `put` 接口，尽管与 `increment` 函数相比，`put` 会存在一定的局限性。

- `put` 需要初始值。比如，增量为 2，你首先需要得到初始值，这个值在我们的例子中为 42。在将值 44 发送回 HBase 之前，需要在客户端上完成加法。
- 因为需要初始值，所以无法在一个多线程任务中增加一个值。

Spark Streaming 这样的数据流方案也许可以消除这种限制。如第 7 章所述，Spark Streaming 能够在一个 RDD 中存储计数，在 Reduce 处理中进行增量，然后将这些值 `put()` 到 HBase 中。尽管很简单，但这也提供了一个强大的模型。我们进行了分区与 Microbatch，因此这里不需要多线程。同样，也不需要由每一个 pass 从 HBase 中 `get` 数据，因为只有一个更新值的系统，所以只要不丢失 RDD，就不需要重新 `get` 的最新值。

总结来说，HBase`increment` 使用时比较简单，但是存在复制增量的问题。`put` 不存在这种问题，但是会持续更新，所以更加复杂。上述例子的实现使用了 `put`。

3. 使用HBase的`put`方法记录事件历史

除了简单的计数事件，我们也想为每个用户存储全部交易历史。由于一个 HBase 记录有其特定的行、列与版本，将用户的新事件放置到与历史数据相同的行与列中，就可以存储用户的历史，并使用 HBase 追踪用户历史。

与 `increment` 相同，HBase 版本化提供了很多功能，但也存在缺陷。

- 每个单元存储的版本号（version）默认值为 1。这种情况下，我们想要将版本号设为一个更高的数值——至少为 20，也许更高。版本号的设置在列族而不是列的范围内进行。所以，如果只想把一行中一列的版本号设置为 20，你需要确定是否真的想让所有列的版本都为 20，或者付出额外的代价拆分出两个列族。
- 如果决定对所有列的版本增加20，那么表的负担就会增加。首先，扫描时间会拉长。第二，进入模块缓存的有用信息量会减少，合并后磁盘上表的大小甚至会增加。

还有其他方法吗？答案是有，而且还是 HBase 的 `put`。客户端可以从 HBase 中读取所有的历史，创建新事务，而后将全部历史写入一个单一的 `put`。

与 `increment` 及 `put` 方法不同，采用这种方法无法轻易作出明确的决策。我们可能想保留最后 100 个 IP 地址。这时 `put` 方法会涉及大量更新。同样，检索这么多历史并存储到本地缓存，需要付出的代价也非常大。在确定模式设计之前，你应该仔细考虑具体情况与两种方法的性能。

现在看一下如何从网络服务器上更新 HBase 中的画像。注意，本例是经过简化的代码片段，完整的应用见 GitHub 仓库（https://github.com/hadooparchitecturebook/hadoop-arch-book）。

在这个例子中，每个画像都有自己的行，行键为 `userId`。行包含两个列：json 存储用户画像，格式为 JSON，时间戳存储画像更新的时间。通常不推荐将画像存储于 JSON 格式中，因为 Avro 格式的优势更显著（如前面章节所述）。在低延迟数据流处理中，JSON 占用的 CPU 更少，能够增加 Web 服务器的吞吐量。

这里的整体计划就是开放与 HBase 之间的联系，初始化一张表，然后从表中 get 与 put 画像。

```
static HConnection hConnection;

hConnection = HConnectionManager.createConnection(hbaseConfig);

byte[] rowKey = HBaseUtils.convertKeyToRowKey("profileCacheTableName", userId);
Get get = new Get(rowKey);

final HTableInterface table = hConnection.getTable("profileCacheTableName");

Result result = table.get(get);  ❶
NavigableMap<byte[], byte[]> userProfile = result
                  .getFamilyMap("profile");


Put put = new Put(rowKey);  ❷
put.add("profile",
        "json",
        Bytes.toBytes(profile.getKey().getJSONObject().toString()));
put.add("profile",
        "timestamp",
        Bytes.toBytes(Long.toString(System.currentTimeMillis())));

long previousTimeStamp = profile.getKey().lastUpdatedTimeStamp;

table.checkAndPut(rowKey,
                  "profile",
                  "timestamp",
                  Bytes.toBytes(Long.toString(previousTimeStamp)),put)  ❸
```

❶ 代码段从 HBase 中获取画像。在完整应用里，它出现在 loadProfileFromHBase() 方法中。在缓存中搜索画像时，如果画像不在缓存中，这段代码就会执行。

❷ 代码段更新 HBase 中的画像。在完整应用里，它出现在 HBaseFlusher 中。HBaseFlusher 是一种背景线程，能从列表中读取更新的画像并将其输入 HBase，如上所示。

❸ 我们使用 checkAndPut() 方法来避免出现竞争条件，防止意外重写已经被另一个 Web 服务器更新的画像。如果 HBase 中的时间戳与 app 中的不符，那么就会出现一个过期的副本。这里需要从 HBase 中得到一个新的副本，对其更新，而且试着重新将其写入 HBase。这行通常出现在一个 while() 声明中，所以如果 checkAndPut() 失败，我们会重新加载新的画像、更新画像并重试。

9.7.3 事务状态更新：通过或否决

现在来讨论发现欺诈时如何将通知发送到系统。

当客户端发现欺诈时，外部系统需要开始行动作出反应。从客户端方面来说，第一个外部系统最可能首先将事件发送到客户端，并要求获知请求或行动是否合法。在简单的架构中，让这种方法警告初始发送人，与针对发送者请求作出反应同样简单。

下一组可能需要知道欺诈检测结果的外部系统在下游，而且不在实时反应窗口之内。我们能将检测事件直接发送到这些系统，但是这几乎等于复制这些系统。进一步来说，如果存在一个以上的下游系统，那么可能要求多次提交同一个客户端警告，这也增加了 NRT 客户端方法的加载量。这种情况下将 Kafka 或消息队列（Message Queue，MQ）系统增加到架构中可能是更好的选择。可以将警告发送到 Kafka 或 MQ 系统中的一个主题，允许所有需要访问警告的下游系统订阅这个主题。

下一步最需要去做的可能是记录并学习欺诈检测决策。这要求我们发送所有的事件、对应的欺诈检测决策（欺诈、未欺诈、欺诈类型）以及从长期存储与批处理方面考虑的 HDFS 中的决策推论。长期来看，这将使我们能够使用类似于 Spark、Impala 与 MapReduce 的引擎执行更深层的数据挖掘。这种长期存储区域与批处理正是之前讨论过的深层处理。我们随后看一下如何执行这种处理。

最后，事件与决策也能发送到一个数据流处理系统，接受实时分钟的数据学习。本章随后也会提到这部分内容。

9.8 数据采集

接下来关注欺诈检测应用的数据采集部分，也就是图 9-3 架构图中高亮标出的一小部分。

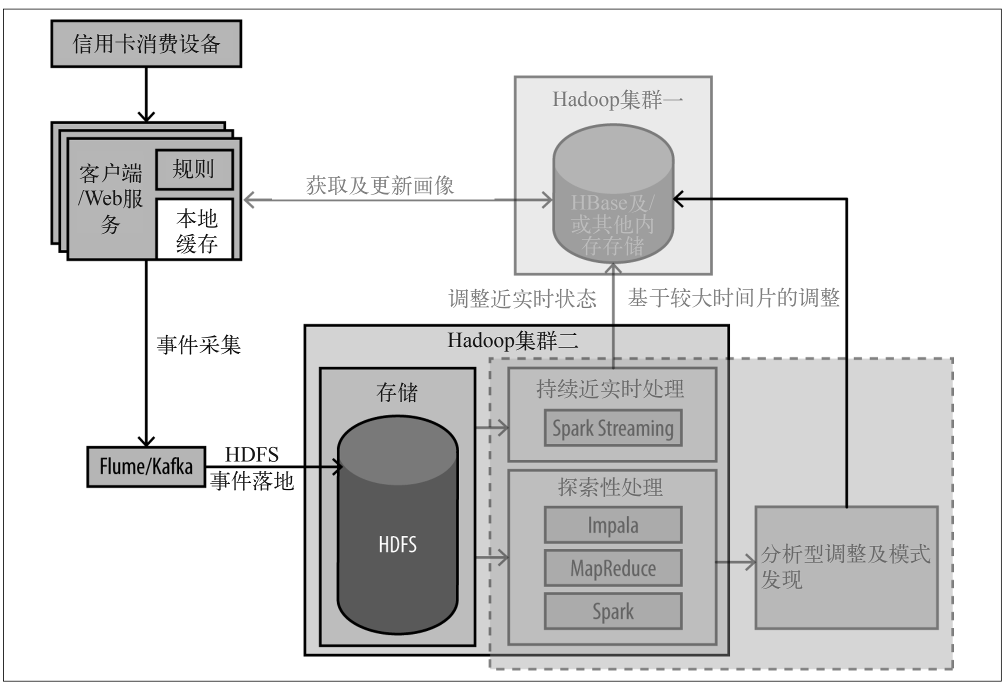


图 9-3：采集架构

这里使用 Flume 作为进入 HDFS 或 Spark Streaming 之前的最后一站，原因在第 2 章已经提到了：配置简单、扩展性好、项目成熟度高。

接下来，我们将关注连接客户端到 Flume，部署 Flume 到 HDFS 的不同方式。关于建设一个 Flume 采集架构，有许多事情要做，大部分相关内容在第 2 章中都曾提及。

客户端与Flume间的数据通路

有很多种方式可以将客户端连接到 Flume。不过，出于讨论的需要，我们只关注以下三种：客户端推送、logfile 拉取以及在客户端与最终 Sink 之间植入消息队列。

1. 客户端推送

第一种方式非常容易上手：在应用中嵌入一个 Flume 客户端进行消息的批处理，并将消息发送给 Flume。在这个模型中，客户端会直接指向 Flume 的 agent，后者会执行写 HDFS 的操作（参见图 9-4）。

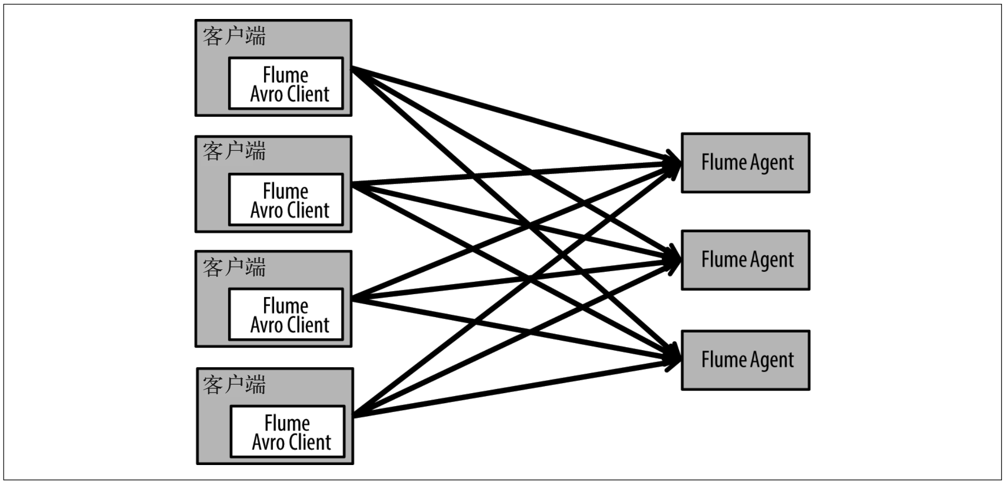


图 9-4：客户端推送

在 Web 服务中，这种方法可以借助 NettyAvroRpcClient 的 API 很便捷地实现，并且在传输过程中的加密、压缩以及对于批处理和线程的细粒度控制上，该接口都可以提供优势。

要想使用 NettyAvroRpcClient，首先需要通过 RpcClientFactory 获取一个实例，然后使用 EventBuilder 创建 Flume 事件，并将事件追加到 NettyAvroRpcClient 中以发送给 Flume。另外，还可以调用 `appendBatch()` 方法一次性地发送一系列事件。这一方法会增加延迟（在发送出去之前要等待事件完成积累），但是能够增加吞吐量，我们推荐使用该方法向 Flume 发送数据。

举例来说，如下是使用 Avro 客户端从欺诈检测应用向 Flume 发送事件的过程（此处只是截取了代码片段，完整的应用代码可以通过访问 GitHub 仓库获得（https://github.com/hadooparchitecturebook/hadoop-arch-book）。

```
public static class FlumeFlusher implements Runnable {

  int flumeHost = 0;
```

```
    @Override
    public void run() {

      NettyAvroRpcClient Client = null;
      while (isRunning) {
        if (Client == null) {
          Client = getClient();  ❶
        }
        List<Event> eventActionList = new ArrayList<Event>();
        List<Action> actionList = new ArrayList<Action>();
        try {
          for (int i = 0; i < MAX_BATCH_SIZE; i++) {
            Action action = pendingFlumeSubmits.poll();  ❷

            if (action == null) {
              break;
            }

            Event event = new SimpleEvent();
            event.setBody(Bytes.toBytes(action.getJSONObject().toString()));
            eventActionList.add(event);
            actionList.add(action);  ❸
          }
          if (eventActionList.size() > 0) {
            Client.appendBatch(eventActionList);  ❹
          }
        } catch (Throwable t) {
          try {
            LOG.error("Problem in HBaseFlusher", t);
            pendingFlumeSubmits.addAll(actionList);  ❺
            actionList.clear();
            Client = null;
          } catch (Throwable t2) {
            LOG.error("Problem in HBaseFlusher when trying to return puts to " +
            + "queue", t2);
          }
        } finally {
          for (Action action: actionList) {
            synchronized (action) {
              action.notify();
            }
          }
        }
      }

      try {
        Thread.sleep(HBASE_PULL_FLUSH_WAIT_TIME);
      } catch (InterruptedException e) {
        LOG.error("Problem in HBaseFlusher", e);
    }
  }
```

❶ 通过传入要发送数据的 Flume agent 的地址，初始化 Avro 客户端。

❷ 维护一个待发送给 Flume 的操作（许可或驳回的决策）队列。从队列中提取事件，并将

它们转化成 Flume 事件。

❸ 将这些事件添加到一个列表中，从而保证以批量的方式发送给 Flume。

❹ 在这里向 Flume 发送事件列表。

❺ 如果有错误发生，那么将操作放到队列中，以便稍后重试。

在初始化客户端时，连接到的 Flume agent 会有一个 Avro 数据源监听对应的端口。

```
a1.sources.r1.channels = c1
a1.sources.r1.type = avro
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = 4243
```

直接从客户端向 Flume 推送数据的缺点如下所示。

- 会假定客户端使用 Java 语言实现。
- 需要将 Flume 的库添加到客户端程序中。
- 需要客户端所在节点上的额外内存和 CPU 资源。
- 当 Flume agent 程序崩溃时，客户端需要支持在 Flume agent 之间进行切换。
- 基于客户端与 Flume agent 以及集群物理部署上的不同，客户端在不同的 Flume agent 上拥有不同的网络延迟。这一点会影响到线程的数量和批处理的大小。

如果你对客户端的实现拥有绝对的控制权（我们在本例中就是这样假设的），而且客户端的主要任务之一是将数据采集到 Flume 中，那么这个做法值得推荐。不过，事情不总是这样的，因此需要考虑其他的方案。

2. Logfile拉取

第二种方式在实际场景中比较常见，这主要是因为它简单易用。大多数程序已经使用了 logfile，因此可以从 logfile 中采集事件到 Flume 中，而不需要更改客户端的代码。这种情况下，我们会将日志写入磁盘，然后使用一个 Flume 数据源将日志记录读取至采集流水线中。图 9-5 展示的就是这种方案的整体架构。

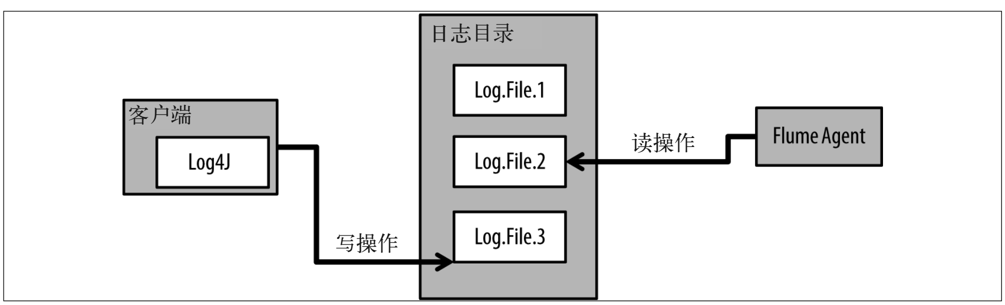


图 9-5：Logfile 拉取

虽然这种方式比较常见。但是我们不推荐在欺诈检测时使用它。与前述的 Avro 客户端相比，使用 logfile 进行采集存在一些缺点。

- 性能

  将日志写入磁盘，然后它们读取到 Flume 中，这样做会引发性能问题。虽然数据的读取

性能可以借助操作系统提供的磁盘缓存提高，但是在写操作、序列化以及反序列化上仍然会有一些损耗。读取跟不上写入的速度，常常会导致更为严重的问题。

- 数据丢失

  对于磁盘上快速更改的日志，获取尾部数据不是一个简单的任务。当文件进行自动更换时，数据丢失的概率也会增加。使用 Spooling Directory Source 能够缓解这一问题，但是这种数据源只是开始处理上层应用写操作完成并关闭的文件，从而增加了采集的延迟。

- Flume 需知晓客户端信息

  Flume 需要知道应用程序的日志位置和文件格式。

在本例中，我们假定客户端是自己编写的，可以使用先前的架构，将 Avro 客户端添加到我们的客户端中。在点击流的例子中，我们假定 Web 服务器是给定的，无法对其进行更改。在这种情况下，使用 Flume 的 Spooling Directory Source 从 logfile 中采集数据。

3. 中间层的消息队列或Kafka

这种模型也非常常见。这一方案将客户端从 Flume 中解耦，允许第三方获得即将到来的消息。根据配置的不同，这一方案的具体情形也有所不同。注意，方案在选择上会相当复杂，我们并没有将其包含到先前提到的架构图中，我们想在图中展示更为直接的方案。

我们的选择可以是：客户端→ MQ → Flume/Spark → HDFS/Spark，如图 9-6 所示。

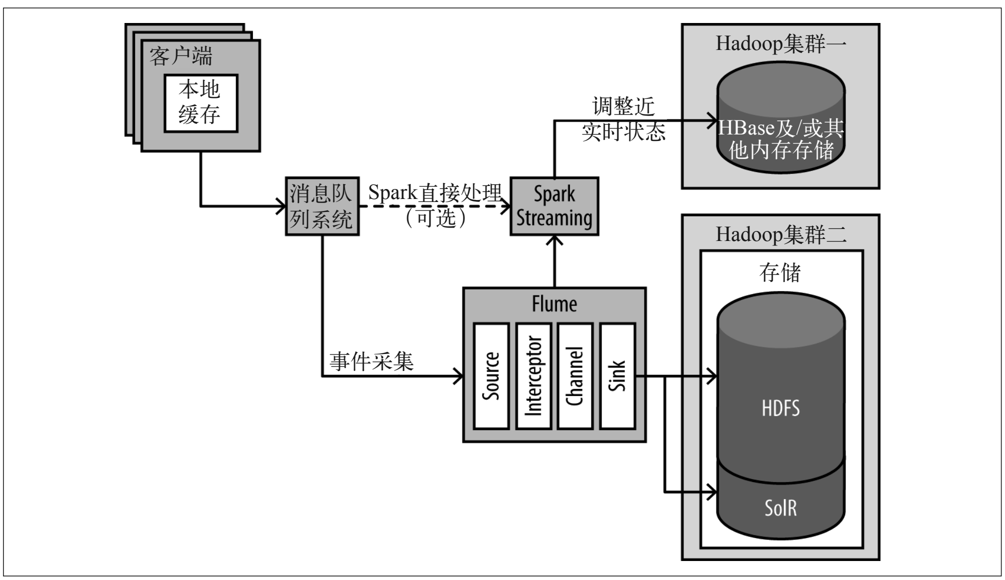


图 9-6：客户端→ MQ → Flume/Spark → HDFS/Spark

这样做的优点在于可以将客户端从 Flume 和 Spark 中解耦出来。注意，图中有两个潜在的数据通路，即 Spark 可以从 Flume，也可以从 MQ 中读取数据。二者均可以正常工作。不过如果 Spark 从 Flume 读取事件数据，那么可以在发送给 Spark 之前进行过滤。这样做可以减少对 Microbatch 系统的压力。

这个架构很简单，集成 Kafka 也是有益处的。先来看一下这些选型，首先是：客户端→ Kafka → Flume/Spark，如图 9-7 所示。

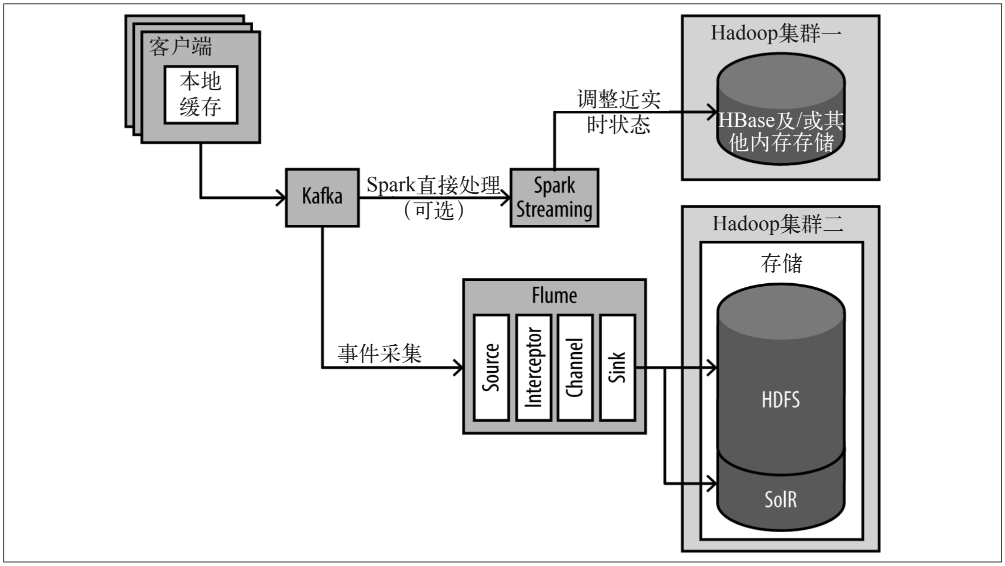


图 9-7：客户端→ Kafka → Flume/Spark

一如前面的例子，这种方式可以在保证功能不变的情况下，从 Flume 和 Spark 中解耦客户端。而且，这样做还有一个很大的益处：可以重放数据，以满足开发和测试的需求。

图 9-8 展示的是另外一种方式：客户端→ Kafka → Flume Sink/Spark。

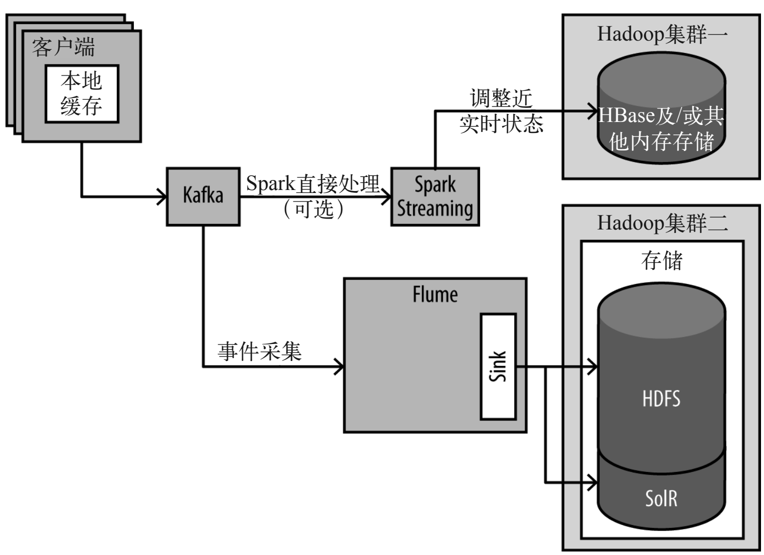


图 9-8：客户端→ Kafka → Flume Sink/Spark

如果你不需要 Flume 拦截器，在数据发送给 HDFS 或 Spark 之前不需要转化或者过滤数据，那么这应该就是最为简洁的方式了。

9.9 近实时处理与探索性分析

正如在整体架构中描述的那样，除了由客户端进行实时决策之外，还需要两类异步数据处理。

- 近实时

  这一类处理通常在事件发生之后的若干秒到几分钟内完成，目的往往是找到涉及多个用户的趋势信息。

- 探索性分析

  这一类处理通常在事件发送后的若干小时或几天内完成，目的往往是分析事件数据、改进欺诈检测算法，或者分析较长时间内的用户行为。这种分析采用的框架通常和数据分析情况类似：MapReduce、Spark 及 Impala 都是比较流行的选择。

正如本章一开始提到的那样，我们会讨论近实时和探索性分析在架构中的可用之处，但不会涉及特定的算法和具体实现。

9.10 近实时处理

迄今为止，我们已经针对欺诈检测遍历了所有的事件，存储了所有决策，并将其发送给 HDFS 或流处理引擎。接下来，让我们使用这些数据，针对欺诈和作弊进行新一轮的防御性检测。

假设已知的作弊行为已经输入到模型和规则库中标记。问题在于总会有聪明人尝试欺诈行为，他们会学习和改进针对他们的模型和规则，根据这些知识进行我们无法想象的策略调整。这就意味着，我们需要一种方式检测出新修改的策略。

我们需要检测出行为的改变。要达到这一目的，不能仅仅在事件级别采取行动，而要着手于更高的层面。如果出现足够多的改变，那么应当有迹象提示我们原始的规则需要调整。更为重要的是，在遭受更为显著的欺诈损失之前，某些活动和操作应当关闭。

想象一下，事件处理或事件级别的验证是针对细节的，是更为专注的处理方式，而近实时处理则试着从更广阔的视角进行观察。这里的问题在于我们需要尽快行动，因此为了看到全局我们会忽略一些细节。我们会关注一些指标（如总量、平均值以及集群的行为），从而了解整体上的行为模式是否超出了原来的预期。

有一个很好的例子可以证明全局事件分析的重要性：一大群人的事件更改会达到阈值。回想我们之前讨论过的欺诈行为的事件验证，一种检查方式是看用户的行为是否在一个全局的或者相对的正常行为阈值之内。以电子产品的正常开销为例，通常情况下，如果一个用户在电子产品上的花费增加了 100 倍，那就可能是发生了值得关注的事情。但是如果这发生在新 iPhone 上市的那天，那么某些从事件级别上看来异常的事件在全局范围看就是正常的了。

同样，我们也可以利用这种方式查找大量的小额诈骗。我们可以查找这样的情况：欺诈行为在单个用户层面上微不足道，但是会影响到许多用户。比如从百万用户各盗取 50 分，这在事件层面上几乎无法察觉，但是在宏观层面上就可以挑选出来了。

近实时分析的一种常见类型是进行窗口分析，也就是关注某一时间窗口（通常是几秒或几分钟）内的事件或事件数量，如图 9-9 所示。

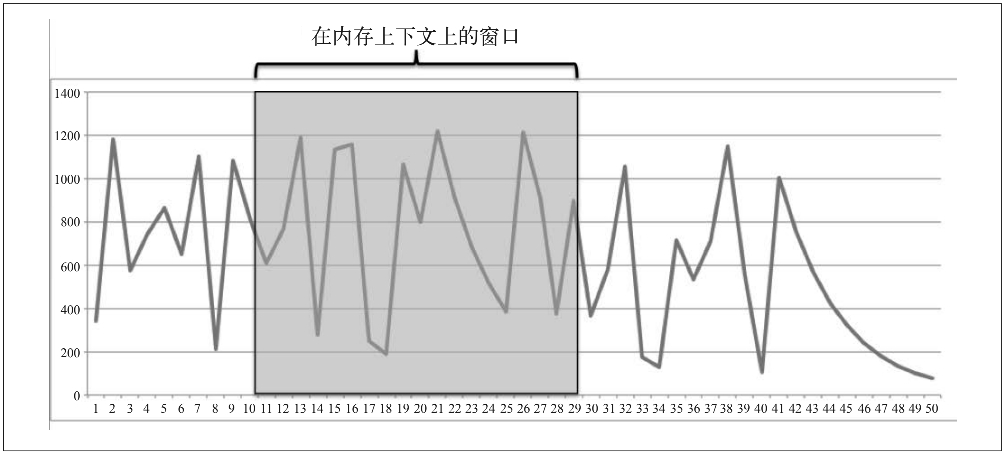


图 9-9：开窗分析

举例来说，窗口分析可以用于检测股票价格的操纵行为。有些模式在事件层次上几乎无法发现问题，然而如果以时间窗口的方式关注股票市场，我们就能看到事件的发生顺序，以及同一时间所有事件的上下文信息。

近实时分析可以通过流处理框架（如 Spark Streaming 和 Storm）或者使用快速批处理框架（如 Spark）进行。

Spark 和 Spark Streaming 之所以能够成为解决方案的候选项，原因如下。

- 我们希望将注意力集中在复杂的业务规则上，不要因为使用流处理的方式实现系统而过度分心。Spark 更高一层的 API 和机器学习库往往很实用。
- 通常情况下，业务规则可以在批处理模式下验证，并在流处理引擎上运行。Spark 支持从批处理到流处理的平滑过渡。
- 最后，我们需要进行许多相当复杂的处理（如持续聚类或推荐），而 Spark Streaming 提供了不少可以让这些处理简化的 API。比如，开窗函数就可使用 Spark 很便捷地实现。

9.11　探索性分析

最后一个要讨论的架构组件就是探索性分析。这一点的实现通常需要数据分析师和欺诈检测专家通过 SQL 和可视化工具在数据中找到有意思的模式。另外，这一步还可以包括使用机器学习算法进行自动模式匹配，以及算法在实时环境下的模型训练。

通常说来，这一部分处理不会拥有和前面处理一样的 SLA 要求。

将事件数据加载（事务处理、通过或否定决策、决策原因）到 HDFS 之后，我们可以通过 SQL 接口将其暴露给非开发人员（如业务分析师）进行即席查询。

这就是分析师在数据仓库进行的工作。在 Hadoop 的帮助下，海量数据的分析比以前更为快速。极短的响应时间能够让分析师迅速浏览数据：运行查询，了解情况，再进行查询。如果分析师“发现情况”，他们可以利用这些知识更新用户的画像，改进实时算法，或修改客户端以获取并利用额外的信息。

分析师可以浏览数据，我们也可以将通用的机器学习算法应用到 HDFS 存储的数据上。一些常用的方法如下。

- *针对打标数据进行监督学习*

  一系列的交易经过打标，可靠地区分成合法或欺诈之后，算法就可以确定欺诈交易是什么样的，因而能够给交易分配风险分值。支持向量机就是实现该方法的常用算法，贝叶斯网络也是。

- *无监督学习*

  举例来说，如 K-Means 这样的聚类算法能够对相似行为进行聚簇，而分析师稍后可以对特定的簇进行欺诈判定。

分析师可以使用 HDFS 上的历史数据来探索新的算法和特征，以便用在已有的算法上，并验证算法和规则。分析的结果可以用于实时和近实时欺诈检测层的改进。

9.12 其他架构对比

当然，其他的架构也可以解决这一问题。我们在此挑选一些方案进行了解，并讨论选择前述架构作为解决方案的原因。图 9-10 所示为目标设计。

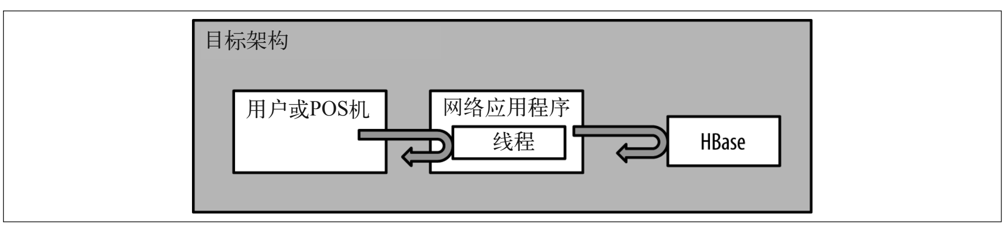


图 9-10：目标架构图

9.12.1 Flume拦截器

正如第 7 章提到的那样，我们可以在 Flume 拦截器中实现交易许可的逻辑，对事件进行处理并在更新 HDFS 的同时更新 HBase 中的画像（如图 9-11 所示）。额外的一层转发能够降低 Web 服务的负载，因为事件的查找活动、HBase 中的画像检查，以及是否许可该交易过程的决定，这些都在 Flume 中完成，而不需在 Web 服务中进行。这样使得服务的扩展性更好。Flume 拦截器能够满足应用低延迟的需求。这样做的缺点在于不太容易将结果返回给发出请求的 Web 服务器，因为已经加了一层的转发。我们不得不通过回调的 URL 或消息系统完成决策结果的返回。

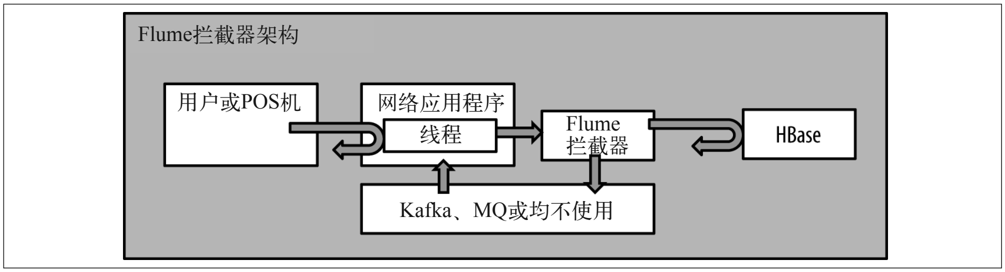


图 9-11：Flume 拦截器架构图

9.12.2　从Kafka到Storm或Spark Streaming

这一架构如图 9-12 所示。与 Flume 拦截器的方案类似，它从 Web 服务中解耦事件处理，并提高了扩展性。在这一架构中，画像的查找、更新以及交易的许可会在 Storm 或 Spark Streaming 的 Worker 中进行。注意，这就意味着，为了进行诈骗检测，在验证事件之前要经过两个系统。要构建一个快速系统的时候，最小化访问路径和复杂度会更有利。而且，这一方案与 Flume 拦截器方案在将结果返回给请求者时，拥有相同的问题。

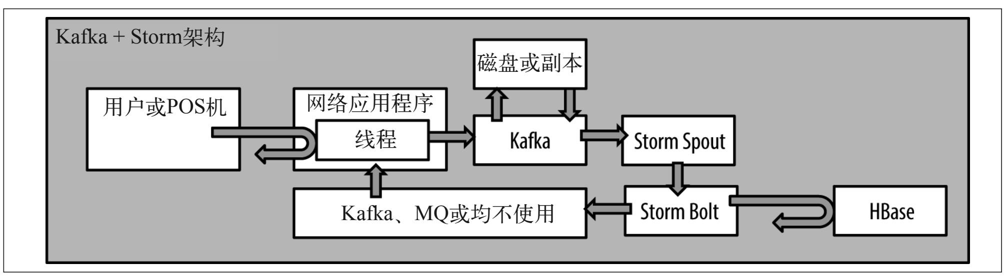


图 9-12：Kafka+Storm 架构图

9.12.3　扩展的业务规则引擎

另外一个方案是使用外置的系统处理业务规则，如图 9-13 所示。客户端会向该系统发送请求，并将获得的结果返回给请求者。这样做的好处在于业务规则引擎能够从应用中解耦，支持作为新信息源的模型更新。这一方案通过引入额外的组件增加了实现的复杂性。无需赘言，业务规则引擎不在本书的讨论范围之内。

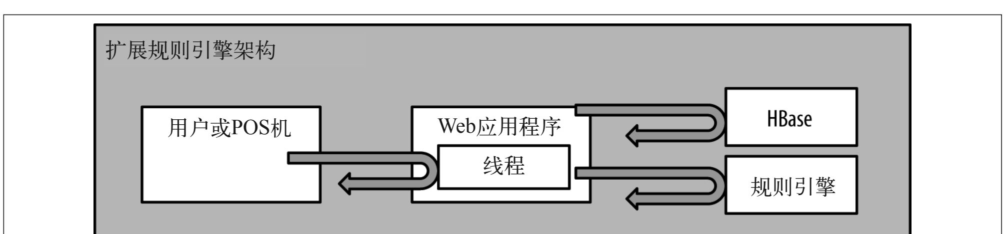


图 9-13：扩展规则引擎架构图

注意，在所有这些可选的架构中，NRT 和探索性分析可以按照之前解释的方式实现。而使用 Spark Streaming 或 Storm 时，有些 NRT 可以利用流处理引擎自身处理。

9.13 小结

实现一个健壮的欺诈检测系统需要对事件作出快速而可靠的响应，并且需要处理大量数据。满足这些需求的架构很难实现，不过 Hadoop 及其生态系统的组件对海量数据的流处理和批处理提供了支持，使用起来获益良多。

针对基于 Hadoop 实现欺诈检测系统，我们描述了一种可能的架构，不过这只是一种实现类似系统的可选方案。在真实场景下，架构会根据具体用例和企业面临的欺诈场景来驱动。好在我们提供了思路，探讨了 Hadoop 如何应用于这种类型的用例，如何应用于需要近实时处理的应用。

第10章 数据仓库

现如今，各行各业都要基于数据作出决策。*数据仓库*也称为*企业级数据仓库*（Enterprise Data Warehouse，EDW），就是一个用于支持数据驱动式决策的大型数据集合存储，已成为各机构数据基础设施的核心。Hadoop 最为常见的应用场景之一就是充当 EDW 架构的补充，通常称为*数据仓库装载*（Data Warehouse Offload，DWO）或*数据仓库优化*（Data Warehouse Optimization，DWO）。

使用 Hadoop 做数据仓库装载，一般涵盖如下内容。

- ETL/ELT

  抽取-转换-加载（Extract-Transform-Load，ETL），即从数据源系统抽取数据，经过转换加工后，将其加载到目标系统中，用于后续处理和分析的过程。转换阶段可以包括基于业务需求的转换、与其他数据源的结合，以及基于某些条件的数据验证或数据过滤。另外一个常见的处理模式是抽取-加载-转换（Extract-Load-Transform，ELT）。ELT 模式先加载数据至目标系统（通常是一组临时表），然后再进行转换处理。关于传统数据仓库领域中 ELT 和 ELT 处理的细节，以及使用 Hadoop 从已有系统进行装载处理的优势，我们后面会进行讨论。

- *数据归档*

  数据增多到一定程度时，传统数据库就很难扩展，而且扩展成本很高，所以通常的做法是将历史数据迁移到外部归档系统（如 Tape）中。这样做的缺点在于，如果要访问归档的数据，就需要将其从线下的归档系统中拉回到线上。也就是说，数据归档会导致拥有潜在价值的数据不在线上，数据的价值无法被发掘。而将数据迁移至 Hadoop 则是高效又节俭的选择，一方面能够像传统数据管理系统（如 EDW）那样导出数据，另一方面还能确保这些数据的在线状态，能够响应分析的需求。

- 探索性分析

  对于并未存储于 EDW 的数据，Hadoop 提供了一个理想的沙盒（sandbox）环境支持分析。这可能是因为数据内在价值不明，可能是因为数据结构特殊或非结构化，也可能只是因为数据量太大，无法加载到 EDW 中。但是，对这类具有潜在价值的数据，Hadoop 能够提供 EDW 所没有的数据访问能力。

许多机构正在利用 Hadoop 来增强已有的数据基础设施。随着最新的进展，Hadoop 已经具备了数据仓库的功能，包括可靠的数据存储以及低延迟的查询响应能力。后者是生成用户报表和运行用户查询所必需的。Hadoop 能够很好地存储和处理半结构化和非结构化数据，并支持与传统结构化数据之间的关联。需要与 EDW 中的数据关联，分析消费者的反馈或社交媒体数据的时候，这一点尤为重要。另外，Hadoop 还支持集成许多流行的第三方 BI 以及可视化工具。

本章示例会展示如何将 Hadoop 作为数据仓库的补充，同时探讨数据仓库传统功能的装载。在深入学习案例之前，我们首先看一个传统数据仓库架构的示例。我们的目标不是全面地介绍数据仓库，而是为案例学习打下基础。

> 关于数据仓库架构的全面介绍，有很多不错的图书可以参考，如《数据仓库工具箱：维度建模权威指南（第 3 版）》。

图 10-1 展示了一个典型 EDW 架构的主要组成部分，包括以下几点内容。

- 操作型数据源系统，通常指存储业务数据的在线事务处理（Online Transactional Processing，OLTP）数据库和操作型数据存储（Operational Data Store，ODS）。举例来说，一个电子商务站点会使用一个 OLTP 数据库存储所有客户访问网站产生的订单、退款和其他事务。这类系统一般会优化单记录查询，从实践角度来讲就是高度规范化以及对模式建立索引。
- 暂存区在作为临时存储区的同时，也是一个进行 ETL/ELT 处理的平台。数据从操作型数据源系统抽取并移至数据暂存区，加载到最终的数据仓库之前在这里进行数据转换。这些转换操作一般包括：数据去重、数据标准化、数据清洗、数据补充，等等。值得注意的是，这里提到的暂存区并非指特定的某个架构，它会随着使用的工具、处理数据的模型等因素而有所变化。举例来说，在 ETL 模型中，暂存区可能是数据仓库之外的一些关系型数据表、平面文件或者是厂商自定义的结构。而在 ELT 模型中，暂存区一般是数据仓库中的一系列表，在这些表上执行转换操作，进而将数据加载到最终表中。注意，无论暂存区的物理实现是怎样的，它仅供数据处理所用，用户不能为了查询、报表等操作访问暂存区的数据。
- 数据仓库是为用户提供数据分析、报表生成等功能的地方。与数据源系统的范式模式设计不同，数据仓库的模式设计是面向数据分析的，适用于多条记录的访问场景。这就意味着，很多时候我们会采用多维模式（如星形模式）设计，这一点我们会在后续示例中加以描述。

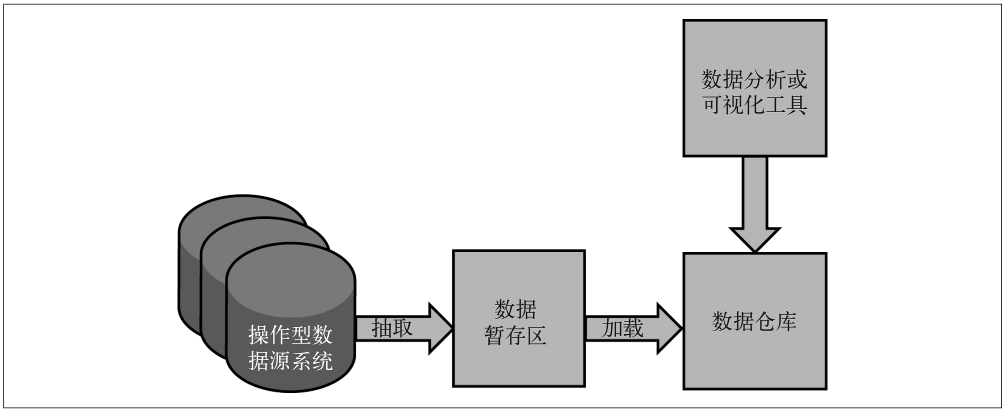


图 10-1：顶级数据仓库架构

以上架构中，多层之间的数据迁移和数据转换，一般是通过数据集成（Data Integration，DI）工具完成的。Informatica、Pentaho、Talend、SAP、IBM 等诸多厂商均提供这种 DI 工具。这些厂商中有不少不仅提供传统的数据管理系统，还提供基于 Hadoop 的解决方案。虽然本章主要关注开源生态系统中的工具，但是已经选用商用 DI 工具的组织机构也可以使用这些厂商提供的 Hadoop 工具。一方面，利用熟悉的工具更容易上手，另一方面，既然已经有了相关投资，那么当然要物尽其用。

10.1　使用Hadoop构建数据仓库

多年以来，服务于机构的传统数据架构健壮性一贯不错，为什么许多组织还要将 Hadoop 集成到数据管理架构中呢？正如你猜测的那样，原因在于不断增长和愈加复杂的数据。以下就是由此带来的挑战。

- 失信的 SLA

  随着数据的量级和复杂度不断增加，数据的处理时间也不断延长。在限定时间内越发难以完成指定的 ETL 处理操作。这就意味着需要对原有的数据仓库进行扩容，否则无法保证先前承诺的服务水平协议（Service Level Agreement，SLA）。

- 不堪重负的数据仓库

  在 ETL 处理尚未完成时，对数据仓库的查询将不得不与 ETL 处理（将数据从原始格式转换成数据仓库中支持查询的格式）竞争资源。这会导致用户的查询响应拉长，用户体验变差，甚至还会使 ETL 处理变慢。

- 高昂的成本

  虽然可以通过增加数据仓库容量或升级暂存系统来解决以上问题，但是这意味着为了额外的硬件和软件许可需要付出另外一笔高昂的费用。而为了控制成本，数据仓库中只存储全量数据一个很小的子集（通常是最近的数据），其他的数据归档存储。此时，如果要查询最近时间窗口之外的全量数据，就得大费周折了。

- 缺乏灵活性

  如果需求发生变更，我们通常需要添加新的报表，或者更新先前的模式描述和报表信息。然而，传统数据仓库在架构上并不具有灵活性，许多组织无法快速响应这种变更。这个问题令这么多组织无法将各种数据源（如社交媒体、图片等数据）与传统结构化数据进行关联。

为了应对以上挑战，我们将目光投向了 Hadoop。而作为数据仓库的补充，Hadoop 能够提供以下帮助。

- 将数据转换操作从已有系统装载到 Hadoop。Hadoop 提供了一个十分经济的数据处理平台。由于 Hadoop 支持海量数据的高效并行处理，所以处理时间和处理性能会有明显的改善，承诺 SLA 的也有了保障。
- 将数据处理移动到 Hadoop 中进行，先前占用数据仓库的资源就可以腾出来为用户查询服务了。

另外，使用 Hadoop 还会进一步获得以下三方面的收益。

- 通过 Hadoop 提供的数据访问接口，分析人员可以探索性地发现数据的潜在价值，而这些正是先前无法发现的。
- 将数据仓库中的数据在线归档到 Hadoop 中，可以释放数据仓库中宝贵的空间和资源，不再需要昂贵的扩容了。
- 具有灵活性的 Hadoop 支持模式演进和半结构化、非结构化数据的处理，这就意味着下流报表或模式发生变化后，快速响应变更成为了可能。

图 10-2 展示了 Hadoop 作为数据仓库补充时的顶级架构图。

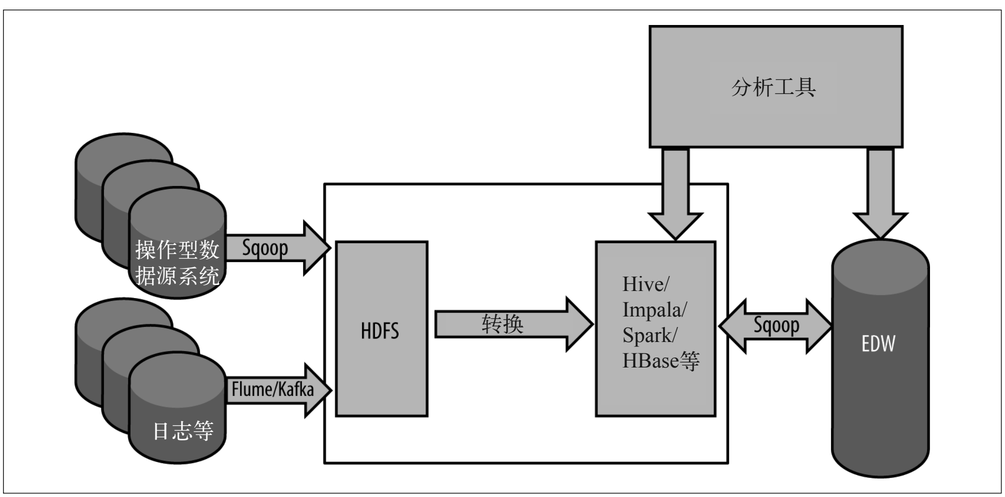


图 10-2：Hadoop 数据仓库架构图

现在来关注整个架构。

- 图中左侧部分是操作型数据源系统（Operational Source System，OSS，如 OLTP 数据库）。通常情况下，除了 OSS，其他数据源也会接入 Hadoop，如网络日志（Web log）、机器数据（machine data）或社交媒体订阅数据（social media feed）等。对于传统架构的数据仓库来讲，这些数据源是难以处理和管理的，但 Hadoop 轻松化解了问题。我们通常会用 Sqoop 完成传统数据源的数据采集，用 Flume 或 Kafka 这样的工具收集基于事件的数据。
- 图中的 Hadoop 已经取代了之前架构图中的数据暂存区。在将数据源转换和加载到目标系统之前，首先要将其加载至 HDFS，而转换操作则通过 Hadoop 支持的数据处理框架（如 MapReduce、Spark、Hive 或者 Pig）来完成。
- 待数据转换操作完成后，数据就可以加载至目标系统了。本例中，转换后的数据要能够通过 Hadoop 上的 SQL 接口（如 Hive 或 Impala）进行访问，还要能够导出至数据仓库，以便后续的深入分析。另外，根据图 10-2，数据仓库中的数据还常常需要回迁到 Hadoop 中，举例来说，对数据进行归档，或者进行探索性分析时就有这样的需要。
- 图 10-2 还展示了通过分析工具（如 BI、分析、可视化工具）对数据进行访问的过程。与先前提到的传统数据仓库形成对比的是，旧架构中的分析工具无法访问暂存区的数据。而在新的架构中，基于数据仓库和 Hadoop 均可以进行数据分析。由于 Hadoop 提供了一个通用的大数据处理框架，在单个集群中支持多种类型的任务（如 ETL 和分析）也是很容易的。
- 图中有一点尚未提及，即处理流程可以通过 Oozie 这样的工作流管理工具进行协调调度。

本章会借助一个案例讲解如何使用 Hadoop 完成数据仓库的装载，主要包括以下内容。

- 使用 Sqoop 从操作型数据源系统中提取数据，并将其导入 Hadoop。
- 将数据存储到 Hadoop 需要考虑的因素，包括数据模型、文件格式，等等。
- 使用基于 Hadoop 的处理工具进行数据转换。
- 使用 Oozie 对整个处理流程进行协调调度。
- 通过 Impala 这样的工具分析 Hadoop 中的数据。

下面介绍例子中需要使用的数据集，并开始具体的讨论。

10.2 用例场景定义

本章将讲解一个电影评价系统。该系统首先包括一张电影列表，包含上映时间、网路电影资料库（Internet Movie Database，IMDb，http://imdb.com）地址等额外的信息。用户登录后可以对电影投票打分，因此评价系统会包含用户的统计信息（如年龄、职业、邮政编码等）。用户给出的分数为 1~5，评价系统存有这些评分数据。每个用户打分过的电影数不定，从 0 到全部均有可能。

这个系统建设好之后，需要回答以下几个问题。

- 一部或多部指定的电影的平均评分是多少？
- 本周的趋势是怎样的？一周中哪部电影的评价分数上升最快？
- 21~30 岁的年轻女性给电影《死囚漫步》（1995）打分不低于 3 分的比例有多大？

- 有百分之几用户会在三个月内修改电影的评分分值？更进一步来讲，电影的评分被修改的比例是多少？对于发生评分更新的电影，更新间隔的时间分布是怎样的？

本例会基于 MovieLens（http://grouplens.org/datasets/movielens/）进行研究。在 MovieLens 的网站上，有十万、百万、千万等几个不同数据量级的数据集，每个数据集的模式和结构都有所不同。我们选择十万量级的数据进行后续的讨论。下载了该数据，你会发现其中有大量文件。其中，我们最为关心的是以下内容。

- 评分数据

  位于名为 u.data 的文件中，包括所有针对电影的评分数据。在该文件中，电影和用户均以 ID 形式简单表示。

- 电影数据

  位于名为 item 的文件中，包含所有电影的 ID，以及对应的其他信息（电影标题、上映日期等）。

- 用户信息

  位于名为 u.data 的文件中，包含所有用户的 ID，以及用户相关的统计信息。

我们将讨论在这样一个电影评价系统背后，相关数据如何通过联合使用 Hadoop 和数据仓库进行存储和处理。随着讨论的推进，我们会首先讲述产生以上数据的系统会有怎样的 OLTP 模式，然后讨论将数据传输到 Hadoop 和数据仓库之后，数据将会拥有什么样的表现形式。

10.3 OLTP模式

数据集是以文件形式存在的，图 10-3 中的实体关系图（Entity Relationship Diagram，ERD）展示了该数据集对应的一个 OLTP 实现。

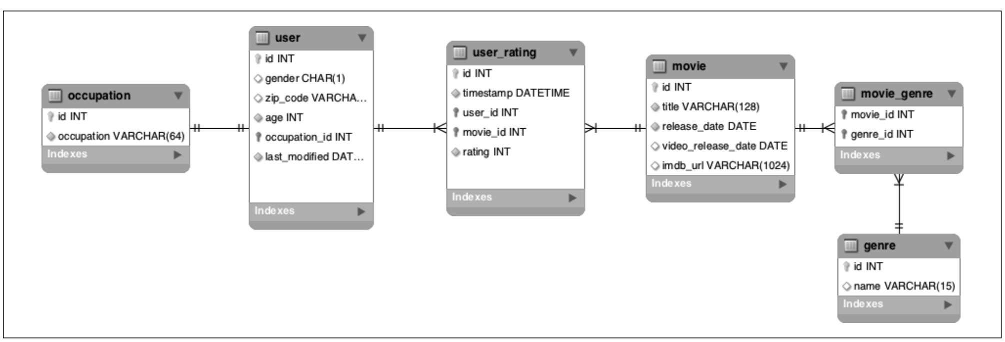


图 10-3：MovieLens 数据集的 OLTP 实现示例

通常来讲，建立 OLTP 模式的目标在于实现数据的规范化，即去除数据冗余、保证数据完整。它能够允许用户更为简便和自动地完成某些操作，如创建、删除、更新特定表中某一字段，同时也能自动地泛化到其他表中。

在图 10-3 中，用户信息表（即 user 表）包含每一个用户的信息，如用户标识 ID、性别、职位、邮政编码以及年龄。而用户的职位信息则保存在另外一张名为 occupation 的职位信息表中。与 MovieLens 原始的模式相比，我们的模式设计进行了一个小更改，在用户信息表中添加了一列，名为 last_modified（上一次修改日期）。该列用来存储用户信息最近一次被修改的时间戳。

你可以看到，电影信息表（即 movie 表）存有所有可以评分的电影。电影信息表中有一个 ID 列，该列是表的主键。表的其他列有：电影名称、上映日期、视频发行日期（若存在）、IMDb 地址。每部电影还对应着一个或多个类别。电影的类别保存在电影类型表（即 genre 表）中。通过电影与类别关系表（即 movie_genre 表），你可以看到每一部电影与一个或多个类别的对应关系。

另外，还有一个用户打分表（即 user_rating 表），其中存储着每个用户对每部电影的打分情况。该表保存着一个用户评分操作发生时，用户 ID（`user_id`）、电影 ID（`movie_id`）以及用户给出的评分、评分操作的时间戳。用户打分表的主键是一个名为 `id` 的列，当新的打分记录插入时，自增的值对应产生。

10.4 数据仓库：术语介绍

数据建模有两种常见策略：*维度模型*（也称为星形模式）与*规范化模型*。传统数据仓库设计通常会涉及以上二者之一的变体。

维度模型始于一张事实表（fact table），这类表的内容通常是数据库中关于实体的不变值或测量值。举例来说，事实可以是电信业务中的一通呼叫、零售电商业务中的一笔订单、医疗保健行业中的一剂处方等。在上述案例中，事实是一个用户针对某一部电影给出的一次评分。

在维度模型中，围绕事实表会有许多维表（dimension table），这些表中是系统中各实体的详细信息，给出了事实表的上下文。在零售电商业务中，某个商品在特定价格的成交订单就是事实表中的一条记录，而维表的维度则会涉及商品、付费的客户、支付日期、支付发生的地理位置，等等。商品维表中有关于这件商品的详细信息，如它的重量、制造商名称、入库时间。在上述例子中，维表就是电影和用户的诸多相关信息，如电影的上映日期、URL，用户的年龄、职位、邮政编码，等等。

对于数据仓库来讲，采用规范化模型的方案则类似于 OLTP 数据库的设计。按照实体创建和设计表，并要求遵从数据库第三范式（Third Normal Form，3NF），即属性全面依赖于主键。在零售电商业务场景中，商品表包含商品名（因其完全依赖于商品 ID），但该表中不应该存在制造商名称（因其不依赖于商品 ID）。于是，基于规范化 DWH（Data Warehouse）模式的查询往往会涉及一连串的关联操作。

一般而言，多维度设计方案能够对分析性查询起到简化和加速的作用，而规范化设计方案更有利于更新操作。

在电影评分的例子中，我们选择采用维度模型的数据仓库，这是因为它更适用于 Hadoop 分析。我们稍后会详细探讨这一点。

现在快速回顾维度设计的基本概念，对基于 Hadoop 设计 ETL 过程或数据仓库而言，这些概念非常实用。

- 粒度

  粒度（grain）是指事实表的粒度大小。事实表中的一条记录代表什么？一个包含多个商品的订单？还是一个大订单中的单个商品？出于一致性分析的要求，事实表中的所有事实记录必须是同一粒度的。粒度应该尽可能地细小，以便用户通过分析数据回答意想不到的问题。在这个例子中，粒度为一个用户对某部电影的评分。

- 可加性

  事实表中的有些字段是可以直接相加的（如以美元为单位的销售额），有些不能相加（如电影的评分）。前者可以进行求和，对于后者通常需要计数或者求平均值。在本例中，评分对应的事实表不满足可加性，但可以求取平均值。

- 聚合事实表

  这些是为了加速查询而构建的细粒度的事实表的聚合。聚合事实表（aggregate fact table）与细粒度事实表（granular fact table）均可以通过 BI 工具进行访问，用户可以根据分析所需的粒度选择合适的表。在本例中，存储每部电影的最新平均分值的表就是这样一张聚合事实表。

- 事实表

  事实表拥有一个或多个外键（Foreign Key，FK），这些外键的值会参照维表。事实表的主键（Primary Key，PK）通常是一系列外键的组合，这些外键能够唯一地确定一条记录。在本例中，列 user_id、movie_id 以及 timestamp 能够唯一地确定一条事实记录。

- 维表

  维表是各种查询约束（如查询 10~23 岁年龄段的用户）以及报表标签的源头。

- 日历维度

  日历维度是一种特殊的维度类型，附加在大多数的事实表和一些维表上，比简单的日期维度更为丰富。在本例中，如果想分析电影评分是否会在假期上升，我们不需要使用 SQL 计算圣诞节假期时间，而是动态地查找日历维度，了解相关的信息。

- 缓慢变化的维度

  有一些维度会随着时间发生变化，如用户地址变迁、制造商倒闭、产品线变更名称，等等。该问题有多种处理方式，最常用的无外乎以下两种：要么把已存在的旧值修改为新的值，要么使用新值添加一条新的记录，并将其标记为当前值，将旧的标记为废弃。前者意味着数据的就地重写，以及相关聚合事实表的重建。后者则不需修改历史和重新聚合表。在本例中，用户数据集也包含缓慢变化中的维度，上文中提及的两种策略均会用到。我们通过更改用户信息表中已存在的值，对数据进行更新。这里还会有一个数据集作为用户历史信息表（即 user_history 表），包含每次更新时的新值。两种策略均会使用，为不同的使用模式提供最佳的访问方式：对用户的最新信息感兴趣则使用用户信息表，对用户的历史信息感兴趣则使用用户历史信息表。

10.5　数据仓库的Hadoop实践

发展到今天，存储在 OLTP 数据库中的数据均可以保存到 Hadoop 中。然而，二者的存储是不同的，与 OLTP 数据库中的数据访问模式相比，Hadoop 上数据的访问模式截然不同。

OLTP 数据库可以快速进行原子级的插入、更新、删除操作，同时还能够保证数据的完整性。同一时刻通常只有一条记录被操作。来自应用程序的一次更新或删除操作会涉及多张表，直接涉及的表越少，越方便使用。因此，数据会以高度规范化的形式存储在 OLTP 数据库中，正如 10.3 节提到的那样。

10.4 节已经谈到，数据仓库中的数据存储可以使用维度模型或者是规范化模型。在本章中，对于 Hadoop 上的数据存储方案，我们选用前者。

10.6　架构设计

图 10-4 所示为 Hadoop 数据仓库的顶层架构设计。

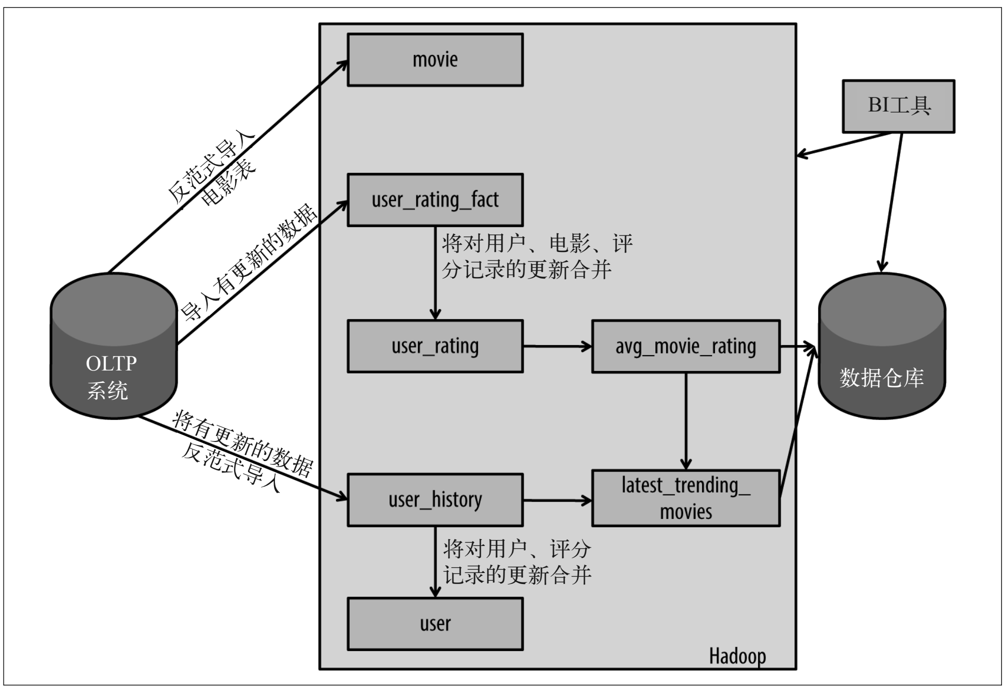


图 10-4：Hadoop 数据仓库的顶层架构设计

从数据仓库的角度来讲，我们需要将三类数据集从 OLTP 数据库导入 Hadoop：可以被评分的电影列表、可以对电影进行评分的用户列表、不同用户对电影的评分。电影数据集的导入任务可以通过 Sqoop 来完成。每次 Sqoop 任务将全量的电影信息导入，覆盖已有的数据集即可。对于用户和打分的数据，要导入的是新增记录和已有记录的更新。这就是所谓

的增量导入，即基于先前导入的数据渐进式地导入数据。这两种数据集均有两种变体：一类是包含所有历史信息、更新时追加记录的数据集（对于打分数据是 user_rating_fact，对于用户数据是 user_history），另一类是更新操作原地生效的数据集（对于打分数据是 user_rating，对于用户数据则是 user）。每次执行 Sqoop 增量导入任务时，首先要更新到包含全部历史信息的数据集，然后将增量合并到原地更新的数据集上。

填充完毕之后，会使用所有这些数据集在 Hadoop 上生成一些聚合数据集（相关知识详见 10.6.4 节），包括所有电影最近的平均得分、用户对某一电影的打分次数，等等。这些聚合产生的表稍后会导出至 EDW 中，供各种工具和不同用户访问。以上所有的处理均可以通过协调调度工具（如 Oozie）进行调配。BI 工具可以直接查询 Hadoop 上未导入数据仓库的全部数据。

接下来，我们基于本例进一步讨论数据建模、数据存储、反向规范化、数据处理以及协调调度的问题。

10.6.1　数据建模及存储

本节讨论如何在 Hadoop 上对 MovieLens 数据集进行建模和存储。

1. 存储引擎的选择

我们首先考虑选择 HDFS 还是 HBase 作为存储。这个问题的答案取决于数据的访问模式（参见第 1 章）。在数据仓库的应用场景中，你可以使用 Hadoop 执行 ETL 工具的一些任务。在大多数数据仓库的实现中，ETL 都要执行批处理任务：将每天的数据从 OLTP 系统中提取出来，进行批量处理，然后批量加载到数据仓库中。使用批处理的方式执行 ETL 主要有两大原因，其一，ETL 处理中的聚合操作原本就需要整个时间周期的数据。其二，批量处理效率更高，缩短了 ETL 的处理时长。

另外，通过 BI 工具下发的聚合类的查询或复杂的分析语句，即需要扫描全表数据执行聚合操作的查询，均可以直接在 Hadoop 上运行。以 MovieLens 数据为例，类似的一个查询就是“统计年龄 21~30 岁的女性，有多少人会给《死囚漫步》（1995）打出不低于 3 分的评分”。执行这条语句需要读取所有年龄在 21~30 岁之间的女性的数据。通常来说，这个数据量即便建立索引并对表进行随机访问，也无法高效地访问数据。

上面提及的这种访问模式非常适合访问存储在 HDFS 中的数据，因为这里需要的不是高性能的随机读取，而是较高的扫描性能。因此，我们将数据仓库的数据存储在 HDFS 上。

2. 反范式设计

正如前面提到的那样，数据导入 Hadoop 之前，通常会进行反向规范化处理。这样做有副作用，会带来数据冗余，但也有好处，可以为分析提速。在类似 Hadoop 这样的系统中，进行多次关联操作并非最佳做法，涉及小的维表时尤其如此。

让我们来看一个例子。在这个例子中，职位表是一张小表，并且几乎不会发生更改。而且，大多数查询只要涉及职位表，势必会查用户信息表。因此我们有两种方案。

- 我们可以在 Hadoop 中分别维护两个数据集，用户信息表和职位表。

- 我们可以对两个表进行反向规范化处理，实际就是摒弃了访问职位表 ID 列（occupation.id）的需求，将职位信息直接嵌入 Hadoop 上的用户信息表中。

我们实际采用后者——将两张表做反向规范化，生成单个用户信息表——因为在 Hadoop 中，用职位的字符串值代替职位 ID，对存储的影响很小。而且这样避免了使用 occupation.id 进行关联，性能提高了很多。

现在，让我们把注意力转向电影信息表（movie）。常与电影信息表一起使用的还有电影类别对照表（movie_genre）和类别表（genre）。这里的类别表与上面的职位表相似，都是较小的维表。而且该表总是会与电影表和电影类别对照表发生关联。对应电影数据来讲，我们也有如下两种方案。

- 我们在 Hadoop 中维护三个数据集：电影信息表、电影类别对照表以及类别表。这样做的好处是可以简化数据采集流水线，直接从 OLTP 数据库中将数据复制过来即可。但这样做也有缺点，查询时总是要关联这三个表才行。
- 我们可以对电影信息表、电影类别对照表以及类别表进行反向规范化处理，生成单个数据集后导入 Hadoop。这样一来，我们就需要在数据采集流水线中添加额外的步骤，或者直接导入到 Hadoop 后执行一个数据表合并的任务。因为所有的数据均在 Hadoop 的同一个数据集中，所以这样做能够让查询处理更为快捷。一部电影可以对应多个类型，因此我们很可能会将一部电影的所有类别存储到衍生数据类型（derived data type）中，比如数组（或者以逗号分隔的列表）。

此处我们选择后者——对电影信息表、电影类别对照表以及类别表进行反向规范化处理，生成 Hadoop 上的单一数据集——这样可以让查询变得简洁、高效。

到此为止，我们已经在 Hadoop 上创建了两个数据集：一个是用户信息表，包含每个用户的所有信息，每条记录有一个字符串类型的字段保存着用户的职位名称，另一个则是电影信息表，包含每部电影的全部信息，每条记录有一个复合类型的字段保存该电影的所有类别信息。接下来，让我们来关注最后一张表：用户打分表。你可能会问应该什么时候对数据进行规范化设计。从技术实现角度，对于用户打分表，我们也有如下两种处理方案。

- 我们拥有三个数据集，包括用户信息表、电影信息表以及用户评分表。许多查询需要借助关联操作才能完成。
- 在 Hadoop 上我们可以只保留一个数据集（即用户评分表），将所有的表进行反向规范化处理（举例来讲就是，一个用户对每部电影的每次评分记录均包含对应电影和该用户的全部信息）。这样一来，不需要借助关联，查询也可以变得更为简单。但是，这种做法会让我们的数据采集流水线变得相当复杂。看起来这么做有些过犹不及了，不是吗？

这次，我们选择前者，放弃进行反向规范化。这里保持三个数据集在 Hadoop 上：用户信息表、电影信息表以及用户评分表。鉴于以下若干原因，我们要让这些数据集彼此独立。

- 将小表反向规范化处理（如合并）成较大的数据集，这样的反向规范化才是合理和划算的。前两次的方案选取也遵循了这样的原则，因此职位合并到了用户信息表，而类别和电影类别对照表合并到了电影信息表。而对于用户评分表来讲，用户信息表和电影信息表的数据量已经相当庞大，执行合并操作将非常耗时，况且查询单个表也节省不了太多时间。

- 如果做了反向规范化处理，信息更新起来将会非常艰难。如果一个用户更新了她的年龄，那么接下来就要对她的每条打分记录进行更新。需要更新的数据极多，这样做很不合理。

> 在使用 Hadoop 的时候，决不能不惜一切代价地避免关联操作，而是应当参考将数据合并到一张表的成本。如果成本不高，那么可以考虑进行反向规范化（如进行预先聚合处理）。而具体的选择则取决于数据模式、数据分布、数据集大小以及使用的工具。关联操作较为费时，部分原因在于没有一种机制能保证两个数据集的对应记录（即满足同一个关联条件的记录）在集群的同一个节点上，而将数据集通过网络进行流式分发是需要花费时间并消耗资源的。将数据合并成一个数据集则保证了一条记录的各种字段（合并前来自不同数据集）存在于同一个节点上。

因此，在我们的架构设计中，Hadoop 上会有三个数据集：用户信息表、电影信息表以及用户评分表。尽管 Hadoop 不是传统意义上的数据仓库，并不适合用 ERD 来描述。图 10-5 展示了 Hadoop 上数据模型的 ERD 描述。要注意的是，职位表（occupation）已经反向规范化到用户信息表（user）中，电影类别对照表（movie_genre）与类别表（genre）已经反向规范化到电影信息表（movie）中。

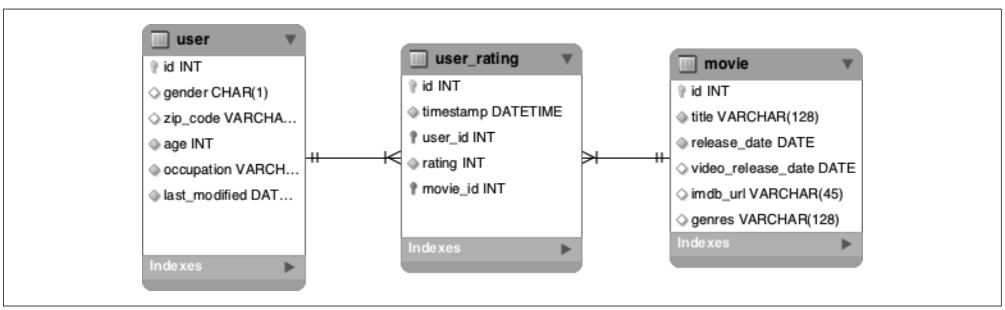


图 10-5：MovieLens 数据集的 Hadoop 数据建模

以上虽然不是最终的架构设计，但已非常接近了，你很快就会明白这一点。

3. Hadoop中的数据更新

本节讨论如何获得数据的更改，并将其存储到 Hadoop 上。这里的 OLTP 模式支持记录级别的更新。举例来说，用户可以更新他的邮政编码，或者更新先前对电影打的分数。Hadoop 上对应的数据模型需要支持这种更新，这一点非常重要。但是，我们不能在 Hadoop 上简单地原地重写更新后的记录，原因如下。

- 我们将数据存储到了 HDFS 上，而 HDFS 是不支持原地更新的。进行更新需要增加一些逻辑上的支持。
- 原地更新后，数据跟踪就无法进行了。举例来说，这里可能会有回看的需求，统计首次评分后三个月内更新评分的用户占比。原地更新后，先前的数据就会丢失，这样的分析也就无法执行了。

让我们逐个讨论支持更新的数据集。用户数据集中有一列名为 last_modified，存储着指定用户上次更新信息的时间。而另外一个支持更新的数据集是用户评分表，包含一个时间戳类型的列，存储着评分分数更新的时刻。如果分数从未更新过，则该字段存储第一次评分发生的时刻。以上两个数据集只要发生更新，对应字段的时间戳字段就会更新。如此一来，如果可以追踪这个字段的更新历史，那么我们也应该能够根据时间将不同的记录更新记下来。

在 Hadoop 上的方案是这样的，我们要创建一张不重写，只进行追加的表，而不是对记录原地重写。拿用户数据集来说，这里要将所有用户不同时间的数据均存储下来。这个只支持追加的表称为 user_history。而对于用户评分数据集，我们将只支持追加的表称为 user_rating_fact，因为这张表代表着一直未被更新、未曾改变的用户评分数据。这里借用了数据仓库中的一个术语：事实（fact）。

数据采集任务的执行过程中，每次都会获取一张新的记录列表，要么是需要插入 Hadoop 数据集的，要么是对已有数据进行更新的。这些新记录会追加到 Hadoop 数据集尾部。

让我们看一个例子：假设 OLTP 数据库中第一天的用户数据集如表 10-1 所示。

表10-1：OLTP数据库中未发生更新的用户信息表

ID	性别	邮政编码	年龄	职位ID	上次更新时间戳
1	M（男）	85711	24	20	1412924400
2	F（女）	94043	53	14	1412924410

Hadoop 上对应的数据集如表 10-2 所示。

表10-2：Hadoop上未发生更新的用户数据集

ID	性别	邮政编码	年龄	职位	上次更新时间戳
1	M（男）	85711	24	技术员	1412924400
2	F（女）	94043	53	其他	1412924410

现在，我们假设第二天用户 ID 为 1 的用户更新了他的邮政编码。另外，这里新增了一个用户 ID 为 3 的用户。那么在第二天，OLTP 数据库更新后的表将如表 10-3 所示。

表10-3：OLTP数据库中更新操作发生后的用户信息表

ID	性别	邮政编码	年龄	职位ID	上次更新时间戳
1	M（男）	94301	24	20	1413014400
2	F（女）	94043	53	14	1412924410
3	M（男）	32067	23	21	1413014420

注意，表中用户 ID 为 1 的用户记录中，邮政编码（zip_code）一列的值已经发生了原地更新，从 85711 变为 94301，而记录上次更新时间戳的列（last_updated）也从 1412924400 变成了 1413014400。还有一点要注意的是，用户 ID 为 3 的用户记录添加到了表中。

在 Hadoop 上，更新后的 user_history 数据集如表 10-4 所示。

表10-4：Hadoop上更新后的用户历史数据集

ID	性别	邮政编码	年龄	职位	上次更新时间戳
1	男	85711	24	技术员	1412924400
2	女	94043	53	其他	1412924410
1	男	94301	24	技术员	1413014400
3	男	32067	23	撰稿人	1413014420

注意，有两条记录添加到了 Hadoop 数据集中。新加的第一条记录代表的更新针对 ID 为 1 的用户，新加的第二条记录则添加了一个用户 ID 为 3 的用户。这种做法与数据库维护事务日志的方式类似。针对数据的全部更新和插入都进行了跟踪和记录。

你可能已经猜到了，以上方式保留了历史信息，但 Hadoop 上的查询却因此变得复杂了许多。举例来说，如果想查询用户 ID 为 1 的用户的邮政编码，那么需要扫描整个数据集，并根据 last_updated 列的值挑选最近一条记录。这样会使查询变慢，写起来也很麻烦。与此同时，我们还想获得一段时间之后的用户表的更新操作以及用户数据集的改变等准确数据。与之类似的是，如果有需求查看指定用户对某部电影的评分，那么就要扫描该电影的所有评分数据，并基于数据中的时间字段挑选最近一次的分数。

对此，我们推荐的解决方案是这样的：Hadoop 对每一个数据集均保存两份副本。对于用户数据集来说，一份副本称为 user_history，在 Hadoop 上按照之前描述的方式存储用户的信息。作为一个只支持追加的数据集，它能够保存用户信息表中的所有更新或新增的行，以供跟踪使用。第二份副本称为 user，为每个用户仅存储一条记录，以提供对最新状态的查询。与之类似，对于评分数据集来讲，第一份副本为 user_rating_fact，存储包括更新在内的所有评分记录，第二份副本为 user_rating，存储每部电影每个用户的最新评分记录。另外一个选择是这样的：我们对其中一份副本不进行完全物化，而是获得基于另一份副本的非物化视图。然而，这种方式的性能会比预先生成物化的数据集差许多。

两类数据集相比较，user_history 与 user_rating_fact 数据集适用于需要了解用户和评分数据变化序列的查询，而 user 和 user_rating 数据集则适用于无需关注用户或评分更改粒度，仅需最新值的查询。同一数据集的两份副本拥有相同的数据模式，只是在存储的数据上拥有不同的语义。

对于 user 或 user_rating 这样允许更新的数据集来讲，涉及对它们的查询时，需要不同的访问模式来访问数据的两份副本，原因有以下两点。

- 查询数据集的当前值。比如希望了解用户当前最新信息的查询。
- 查询历史值。比如有查询想要了解用户是怎样修改职位或邮政编码的。

我们假定访问当前值的查询比访问历史值的查询要多很多。典型的数据仓库不会为了加速访问而另外维护表的一份副本，对于 user_rating_fact 这样的事实表尤其如此。这是因为存储空间十分珍贵。然而在 Hadoop 中，存储空间没有那么珍贵。通过数据集的重复加速大多数查询，这样的投资非常划算。

对于用户数据和评分数据分别维护两个数据集，相关工作流见图 10-6。

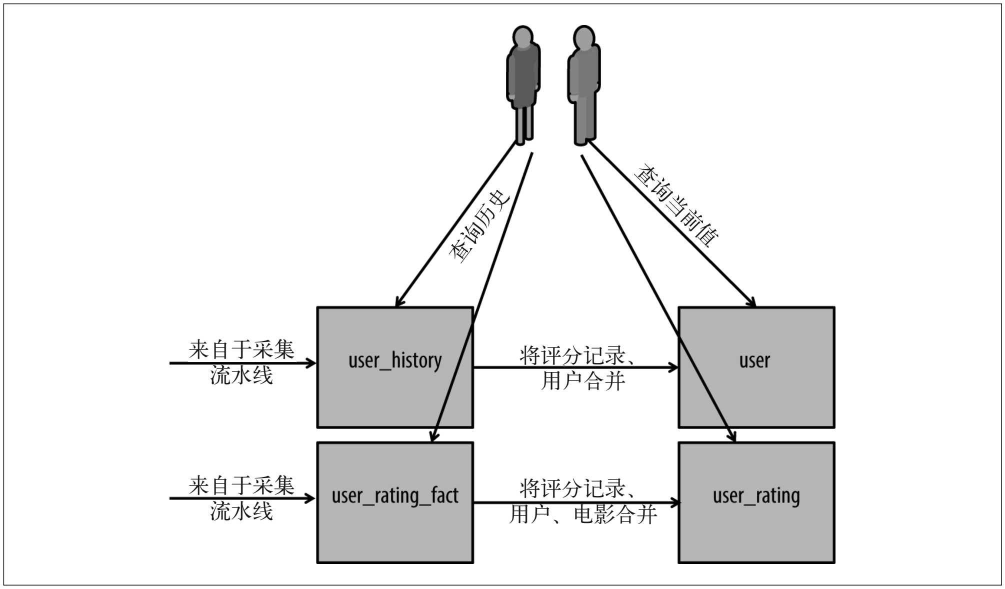


图 10-6：用户和评分数据，每个都有两个数据集的工作流

数据采集流水线在每次执行时，都会将新的插入或更新应用到 user_history 和 user_rating_fact 数据集上。接下来，单独处理的流程将会触发合并任务，进而将新的插入或更新合并到已存在的 user 和 user_rating 数据集上，由此分别产生两个已经执行过插入或更新的新数据集。想进一步了解合并，请参阅第 4 章。

注意，在 OLTP 模式中，电影信息表并没有一个 last_updated 或类似的列。这表明此处对于该数据不支持更新，即使支持更新，也不会去跟踪变化。因此，我们不会针对 Hadoop 中的电影数据集跟踪更新。对于 OLTP 数据库中新增的电影，我们只是在 Hadoop 中将它们追加到电影数据集的尾部。

在 Hadoop 上，我们最终采用的模式如图 10-7 所示。

4. 存储格式及压缩算法的选择

现在我们需要确定，对于 HDFS 的数据集，应当采用哪种文件格式和压缩算法。关系型数据通常是表中的结构化数据，CSV、Avro 和 Parquet 显然都是我们的候选格式。正如第 1 章提到的那样，从性能、压缩和模式管理方面考虑，CSV 不是最佳选择。这些对于数据仓库而言都是至关重要的。在 Avro 和 Parquet 之间进行选择，我们需要好好思索一番。二者都是包含数据描述模式的二进制格式。

Avro 的优点之一在于支持模式演进。这就意味着当 OLTP 模式发生改变时，我们的 ETL 操作不会被打断。ETL 开发者和管理人员都需要关注这个问题：如果模式维护引入不恰当的模式，那么 ETL 操作有可能因此失败。上面提到的优点相当重要。

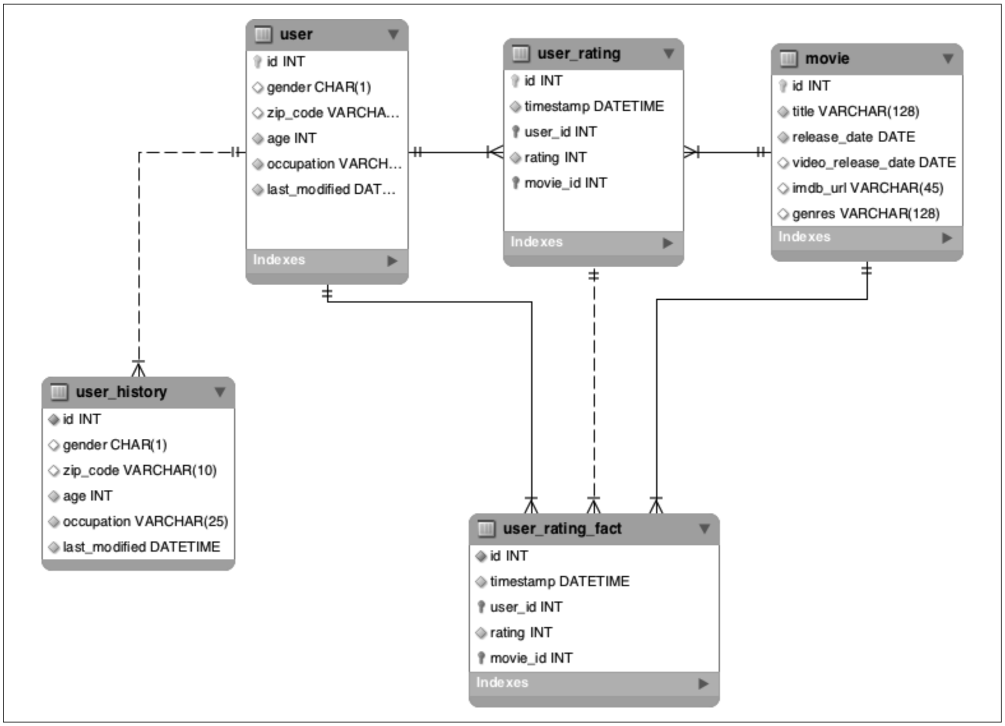
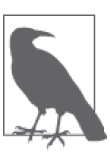


图 10-7：MovieLens 数据集的 Hadoop 数据模型

而 Parquet 格式恰恰有着明显的性能优势。数据仓库中的事实表通常比较宽，这是因为数据会与许多维表相关。而每个分析性的查询或聚合操作通常只会涉及一小部分列。这里提到的查询针对 21~30 岁的女性用户，因此不应访问任何记录的电影类型字段。作为一种列式存储格式，Parquet 允许跳过无用数据所在的数据块。另外，Parquet 表提供了极佳的压缩支持，可以显著地节省存储空间。

选取文件格式的时候，我们需要基于对应数据集的访问方式进行考虑。通常的规则是，行优先存储选用 Avro，列优先存储选用 Parquet。

对于需要支持追加的数据集（即 user_history 和 user_rating_fact），我们推荐使用 Avro。这些数据集通过采集流水线产生，之后需要合并到 user 和 user_rating 数据集上。对于这些数据集来说，执行合并只需获取上次合并与本次合并之间发生的更新和插入数据即可。而且，对于这些更新或插入，需要访问的是整条记录。由此可见，使用 Avro 存储支持追加的数据集是很有道理的。



对于存储在 HDFS 上的 Parquet 数据，推荐的数据块大小为 256 MB。因此，如果 user 数据集压缩后的大小并非比推荐的数据块大小高一个数量级，那么还是用 Avro 存储 user 数据集更好一些。

对于 user_rating 数据集，我们推荐使用 Parquet 格式，因为这里有一张比较大的事实表。

该表的数据量会快速增加，而常见的查询只涉及一小部分列，这种应用场景非常适合采用 Parquet。

对于 user 和 movie 数据集，我们同样推荐使用 Parquet 格式。这是因为针对这些数据集的聚合查询也只涉及一小部分列。举例来说，如果要回答有多少年龄在 21~30 岁的女性用户给《死囚漫步》（1995）的评分不低于 3 分，那么我们需要查询整个用户数据集，但分析所需要的列只涉及用户 ID 和年龄。

至于压缩方式，如第 1 章提到的那样，Snappy 是最常用的压缩编码之一。无论数据集究竟使用 Avro 还是 Parquet 格式，我们都推荐使用 Snappy 压缩算法进行压缩。

总结一下，表 10-5 列出了不同数据集的存储格式和压缩方式。数据仓库中频繁进行小部分列查询的数据集最好使用列式存储格式，因此我们采用 Parquet。支持数据追加的数据集——user_rating_fact 和 user_history——经常会在 ETL 流水线中使用（尽管直接查询也会访问到它们），以生成对应的合并后的表。因此，对于这种行使用方式较多的数据集，我们采用 Avro 格式。

表10-5：不同数据集的存储格式

数据集	存储格式	压缩算法
movie	Parquet	Snappy
user_history	Avro	Snappy
user	Parquet	Snappy
user_rating_fact	Avro	Snappy
user_rating	Parquet	Snappy

5. 分区

接下来，我们需要考虑的是是否应该对 Hadoop 上的数据集进行分区。如果需要，应该按照哪一列进行分区。正确的分区方式能够减少大量不必要的 I/O 操作，因为执行该查询的执行引擎可以直接跳过不必要的数据。

要想正确进行分区，你需要了解数据访问模式。关于分区，这里总共有如下五个数据集需要考虑：movie、user、user_history、user_rating 以及 user_rating_fact。

我们首先关注支持追加的数据集：user_history 与 user_rating_fact。其数据的产生依赖于采集流水线每次运行获取的新的更新和插入记录。然而，通常情况下，查询只是想知道自上次运行到现在，有哪些更新和插入添加到了这些表上。我们只要最近的结果，并不想查询历史数据，因此如何对它们进行分区显得至关重要。我们按天对这些数据集进行分区。（这里假设有足够多的数据让分区的数据大小合理，尤其是对于 user_history 来说。）根据数据量的大小，分区粒度可以进行调整。

通常情况下，分区的平均大小至少要达到 HDFS 数据块大小的若干倍。根据前面的分析，通常 HDFS 的数据块大小应该达到 64 MB 或 128 MB，那么分区平均大小的典型值应该是 1 GB。如果每天评分的数据比 1 GB 多很多，那么可以按照小时进行分区。从另外一个角度来讲，如果用户新增或更新数远小于每天 1 GB 的数据量，那么可以按照周或月进行分区。

现在，让我们关注终端用户会查询到的另外三个数据集：movie、user 和 user_rating。

以电影数据集为例，我们很有可能对新电影的评分更感兴趣，所以将电影数据集按照电影的上映日期进行分区是有意义的。然而，假设每年新发布的电影有 1 万到 2 万部，那么某一年的电影数据集大小最多就是几百 MB 的数量级。（假设每部电影的平均记录大小为 10 KB，乘以 1 万部，共计 100 MB。）这个数据量对于一个分区来说太少。所以，就电影数据集的数据量来讲，我们不建议对其进行分区。

接下来的问题是，要不要对用户数据集进行分区，如果需要，那么应该根据什么来进行分区。用户信息表数据量较小，我们不对该数据集进行分区。

接下来需要关注的是最后一个，也是存在争议的最大数据集，user_rating。我们有以下几个可选方案。

- 不对 user_rating 进行分区。这就意味着每次查询都要针对整个数据集进行，这并不是最优的做法。另外，如果不进行分区，那么这张表就无法按照分区更新，而是需要重写整个数据集。这个方案会带来 I/O 负载，无论是读还是写的时间都会过分拉长。考虑到这张表的数据量，这个方案就更不合适了。
- 我们可以根据每个评分首次产生的时间戳，按照日期对 user_rating 进行分区。如果大部分查询都基于评分首次产生的日期进行（如统计过去七天的新增评分数），那么按照该日期进行分区是有意义的。然而，在每次进行合并处理时，如果从上次合并到现在有用户更新了自己的评分，那么老的分区就需要更新了。
- 我们可以根据每个评分最近更新的时间戳，按照日期对 user_rating 进行分区。如果大部分查询都基于评分最近更新的日期进行（如统计过去七天全部电影的平均评分分数），那么按照该日期进行分区是有意义的。如果两次合并间发生了更新，那么这里仍然需要更新老的分区。这是因为这张表对于每个用户每部电影只包含一条记录，而这条记录存储着该用户对该电影打出的最近评分值。一位用户更新了对某一部电影的评分，电影重新获得评分，那么对应的记录就需要从旧的分区中移出，移到最近的时间分区中。

以上两种方式选择起来并不容易。决定选择哪一种取决于数据在大多数情况下如何访问。在本例中，我们认为评分数据通常会按照上次更新时间进行访问。因此，我们选择最后一种方式：根据每个评分上次更新的时间戳，按照日期对 user_rating 进行分区。

总结一下，我们将全部数据集的分区策略列在表 10-6 中。

表10-6：各个数据集的分区策略

数据集	分区与否	分区列	备注
movie	不分区	不存在	小表不必分区
user_history	按天分区	用户信息更新的时间戳	以减少使用该数据集生成 user 数据集的 I/O
user	不分区	不存在	小表不必分区
user_rating_fact	按天分区	用户更新电影评分的时间戳	以减少使用该数据集生成 user_rating 数据集的 I/O
user_rating	按天分区	用户更新电影评分的时间戳	以减少基于用户最新评分数据进行查询的 I/O

因为以上五个数据集均可以被最终用户所访问，所以这些数据集都会存放在 /data 目录（更准确地说是 /data/movielens 目录）下。

10.6.2 数据采集

在传统的 ETL 处理过程中，数据从 OLTP 数据库中提取出来，并加载到数据仓库中。因此，我们也希望大部分数据来自 OLTP 数据存储。本节关注数据从 OLTP 数据存储采集至 Hadoop 的具体细节。另外，相关的非关系型数据可以加载到 Hadoop 中，还可以集成到我们的数据分析里。这里的非关系型数据包括来自网站的影评、来自 Twitter 的短评等。

为了进行举例说明，本节关注关系型数据的采集。第 8 章和第 9 章涵盖了从流式数据源（如网络日志和信用卡支付数据）中采集数据的内容。

从关系数据库到 Hadoop 有多种方式可以完成数据的采集任务，Sqoop 是迄今为止最为流行的一种，本章将主要关注这个工具。我们在第 2 章中讨论了 Sqoop 的工作原理，并分享了一些使用方面的小窍门。本章主要探讨 Sqoop 在特定场景下如何使用。另外，这里还可以使用 Hadoop 集成的传统 ETL 工具，如 Informatica 或 Pentaho。数据采集系统（如 Oracle Golden Gate）的调整能够高效地对频繁更新的数据表进行复制。

一些 Hadoop 用户会采取另外一种方式：从关系型数据库将数据导出成文件，再将文件加载到 Hadoop 中。如果原本就有从 OLTP 系统进行每日数据导出的处理，那么可对此复用，将数据加载到 Hadoop。不过如果原本并没有这样的处理，也用不着添加。Sqoop 本身就支持数据导出工具（如 mysqldump 或 Teradata 快速导出工具）完成数据的导入，而这种导入经过了优化，更容易使用，而且久经测试。所以，如果是从零开始，我们推荐使用 Sqoop。

选择 Sqoop 作为导入工具后，我们进一步了解数据导入的细节问题。

有以下几种类型的数据表需要导入。

- *数据几乎不变的表*

  我们可以将这些数据表一次性地导入 Hadoop，导入完成后，可以按需执行重复导入操作。在我们的例子中，所有的维表均常态化地发生修改，比如用户会修改自己的属性，而新电影会上映。所以在本例中，没有哪张表属于这个类型。

- *数据频繁更新的小表*

  我们可以将这些数据表每天导入 Hadoop 一次。由于数据量较少，这里不必担心对数据更改的跟踪，也不用担心导入对可用带宽的影响。在这个例子中，电影信息表数据量较小，因此对应的电影数据集属于这一类。

- *数据频繁更新且无法每天全量提取的大表*

  对于这种表，我们需要确定每天有哪些数据发生更改，并将这些更改应用到 Hadoop 上。这些表可以只支持追加而不支持更新。在这种情况下，我们只需将新的记录添加到 Hadoop 的表中即可。这些表也可能是支持更新的，此时我们就需要对更新进行合并。而 user_rating_fact 与 user_history 均属于这一类型。

上述前两类表通常情况下是维表，不过并不是所有的维表都属于这两类。毕竟它们的数据

量未必像第二类表那样小，更新也未必像第一类表那样少。这些表的数据抽取方式基本相同，不同之处仅在于调度 Sqoop 任务的频率有所差异。

注意，因为这些表的列数不多，数据量也不大，所以我们不会将它们导入成 Parquet 格式，也不会将它们导入成 CSV。考虑到访问数据时要避免数据解析，我们选择 Avro 格式。当然对于这些小表，我们不需要考虑分区的问题。

现在，让我们看一个使用 Sqoop 将电影数据集从 OLTP 导入 Hadoop 的例子。电影数据集数据量不大。因此每次导入时我们只需将数据从 OLTP 数据库复制到 Hadoop 中。不过，值得注意的是该数据集在 OLTP 和 Hadoop 中的模式描述是不同的。尤其是在 Hadoop 上，类别表（gen_re）与电影类别对照表（movie_genre）在反向规范化处理之后，成为了电影数据集的一部分。我们需要执行关联操作，将 movie、movie_genre 与 genre 表合并成一个数据集。对此，我们有两种方式可以采用。

- 我们可以在 Sqoop 任务中执行该关联操作，这就意味着通过 OLTP 数据库执行关联。数据在 Hadoop 落地时已经完成了反向规范化处理。
- 我们可以在 Hadoop 上执行该关联操作，将 movie、movie_genre 以及 genre 表从 OLTP 中原样复制到 Hadoop 中即可。接下来执行 Hadoop 任务，实现三张数据表到单个数据集的反向规范化处理。

问题是选择哪一种方式。一般来讲，Sqoop 支持自定义形式的查询式导入。举例来说，Sqoop 的导入任务可以包含任意的 SQL，具体到这个例子就是 `join` 语句。然而，这种导入仅限简单查询语句，不支持自定义的投影，`where` 子句中不能包含 `OR` 条件。在这种情况下，我们还需要指定额外的参数（如 `--split-by`）来保证查询的并行化。不过这样做的确避免了编写和维护后续 Hadoop 关联任务的麻烦，虽然这个关联任务极为简单。从另外一个角度来说，有一些 OLTP 系统可能在资源上较为紧缺，使用 Sqoop 来执行关联操作会给 OLTP 系统带来风险。我们的建议如下：如果在源数据库上执行自定义形式的 SQL 查询比较简单，且 OLTP 数据库能够安全地执行这样的语句，那么就采用自定义查询的 Sqoop 导入；如果 SQL 查询复杂或者 OLTP 系统资源有限，那么就选择后续运行 Hadoop 任务。

本例中的关联较为简单，因此在 Sqoop 任务中直接指定。

```
#!/bin/bash
# 所有的节点都应拥有访问源数据库的权限

# 如有必要则执行清理操作。即使rm命令执行失败，
# 由于后面有|| :,脚本也能继续。
sudo -u hdfs hadoop fs -rm -r /data/movielens/movie || :

sqoop import --connect jdbc:mysql://mysql_server:3306/movielens
 --username myuser --password mypass  --query \
'SELECT movie.*, group_concat(genre.name)
FROM movie
JOIN movie_genre ON (movie.id = movie_genre.movie_id)
JOIN genre ON (movie_genre.genre_id = genre.id)
WHERE ${CONDITIONS}
GROUP BY movie.id' \
--split-by movie.id --as-avrodatafile --target-dir /data/movielens/movie
```

以上命令在 OLTP 数据库上执行一个关联操作，将反向规范化后的电影数据集导入到了 Hadoop 上。导入后的数据集以 Avro 的格式存放在 HDFS 的 /data/movielens/movie 目录下。

导入完成后，我们需要为其创建一个匹配的 Hive 表，后面我们会使用该表进行转换操作。这就需要从刚采集的文件中拿到 Avro 的模式描述，并使用它建立 Hive 表。

```
#!/bin/bash
# 如果指定模式文件存在,删除之
# 行尾的||:确保指定文件不存在时,脚本仍正常执行
sudo -u hdfs hadoop fs -rm /metadata/movielens/movie/movie.avsc || :
hadoop jar ${AVRO_HOME}/avro-tools.jar getschema \
/data/movielens/movie/part-m-00000.avro | \
hadoop fs -put - /metadata/movielens/movie/movie.avsc

hive -e "CREATE EXTERNAL TABLE IF NOT EXISTS movie
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS
INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/data/movielens/movie'
TBLPROPERTIES ('avro.schema.url'='/metadata/movielens/movie/movie.avsc')"
```

注意，我们通常会将表创建为外表（`EXTERNAL`）。这样做在性能上没有损失，而且我们在删除和修改表的时候，不会丢失任何数据。另外，注意在前面的 `CREATE TABLE` 语句中，我们在创建 Hive 表的时候，没有指定模式。取而代之的是，我们使用 avro.schema.url 属性指定了 Avro 模式，这样 Hive 的模式就可以根据 Avro 模式衍生出来。通过这种方式，我们不必维护数据的两种不同的模式，一个在 Avro 中，一个在 Hive 里。它们可以共用 Avro 中定义的同一个模式。

之前提到的第三类数据表有以下约束。

- 首先，我们需要提取整个表的数据，然后在 Hive 中创建一张带分区的表。
- 然后，我们每天都需要提取出最新的修改记录，将这些修改追加到 Hive 表中，或者合并到对应的表上。

Sqoop 支持增量导入。这就意味着，只要有一列能够标识这一行是否包含有新的数据（无论是时间戳还是自增 ID），Sqoop 就可以将之前一次执行对应的值存储下来，在下一次执行时使用这个值，以保证只获取最新的或者是修改后的行。

为了使用这个特性，我们需要创建 Sqoop 的元数据存储（metastore），它用来存储 Sqoop 任务的状态。如何启动 Sqoop 的元数据存储服务，这取决于 Sqoop 的安装方式。如果使用的是安装包，那么可以使用如下 `service` 命令。

```
sudo service sqoop-metastore start
```

如果你用压缩包安装了 Sqoop，则可以使用如下命令。

```
mkdir -p /var/log/sqoop
nohup "/usr/bin/sqoop" metastore > /var/log/sqoop/metastore.log 2>&1 &
```

如果你使用了管理工具，则应该可以通过该工具启动 Sqoop 的元数据服务。

启动元数据存储服务后，你可以使用如下方式运行增量导入任务。

```
#!/bin/bash
SQOOP_METASTORE_HOST=localhost
# 删除已存在的任务。如果不存在，
# 行尾的||:确保返回成功,脚本继续执行
sqoop job --delete user_rating_import --meta-connect \
jdbc:hsqldb:hsql://${SQOOP_METASTORE_HOST}:16000/sqoop || :

# TODO: Made minor changes. Test this
# 如果Hive不存在于/usr/lib/hive目录下,则需显式指定HIVE_HOME的位置
# 如果使用Apache Sqoop 1.4.6及以上版本,则无须这样做
sqoop job --create user_rating_import --meta-connect \
jdbc:hsqldb:hsql://${SQOOP_METASTORE_HOST}:16000/sqoop \
-- import --connect jdbc:mysql://mysql_server:3306/movielens \
--username myuser \
--password-file hdfs://${NAMENODE_HOST}/metadata/movielens/.passwordfile \
--table user_rating -m 8 --incremental lastmodified \
--check-column timestamp --append --as-parquetfile --hive-import \
--warehouse-dir /data/movielens --hive-table user_rating_fact
```

以上脚本创建了一个名为 user_rating_import 的任务，并将该任务存储到 Sqoop 元数据存储服务中。该任务将根据命令行中的用户名以及 HDFS 上的密码文件连接到 MySQL OLTP 数据库，并将数据库中的 user_rating 表导入 Hadoop。该任务基于上次导入的时间戳字段的最新值执行增量导入。新导入的数据会追加到 Hive 表中，而且以 Parquet 格式存储。后续的每一次执行均只抽取比上一次执行时间戳更大的行，也就是自上一次执行新增或更新的评分数据。

执行以下命令，运行该任务。

```
sqoop job -exec user_rating_import --meta-connect \
jdbc:hsqldb:hsql://${SQOOP_METASTORE_HOST}:16000/sqoop
```

如果原来的表只支持插入，那么导入就完成了。每次运行该任务，Sqoop 任务都会将新的数据追加到名为 user_rating_fact 的 Hive 表中。然而，后续还有一些额外需要执行的任务。

- 根据评分的最近修改时间，对事实表进行分区。
- 如果在应用场景中，表中的数据可以修改而非只能新增（比如用户信息表），那么我们需要将新的更新记录合并到已有的表中。本例需要通过后续的 Hadoop 任务将追加到 user_rating_fact 的数据再合并到 user_rating 数据集中。关于这些任务的详细信息，请参阅 10.6.3 节。
- 我们可能还需要预先执行一些聚合操作，以保证后续的报表能够快速生成。

10.6.3　数据处理及访问

讨论了如何在 Hadoop 上对数据集进行建模和采集，接下来我们来看看如何处理落地存储的数据。

1. 分区

需要数据分区的情况有以下三种。

- 需要对已存在的表进行分区。
- 需要将数据加载到表的多个分区中。
- 需要将导入某个目录的数据作为新的分区添加到表中。

要对已存在的表进行分区，我们首先要创建一个新的带分区的表，然后从旧的表中读取数据，并将数据加载到新表正确的分区中。上面提到的最后一步与加载数据到已存在的分区表做法相同。

我们首先创建一张带分区的 user_rating 表。此表按照年、月、日进行分区，在其他方面与原来的 user_rating 均保持一致。

```
#!/bin/bash
sudo -u hdfs hadoop fs -mkdir -p  /data/movielens/user_rating_part
sudo -u hdfs hadoop fs -chown -R ${USER}: /data/movielens/user_rating_part
hive -e "CREATE EXTERNAL TABLE IF NOT EXISTS user_rating_part(
  user_id INT,
  movie_id INT,
  rating INT,
  last_modified TIMESTAMP)
PARTITIONED BY (year INT, month INT, day INT)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION '/data/movielens/user_rating_part'"
```

接下来，我们将评分数据加载到新表的对应分区中。

```
#!/bin/bash
hive -e "
SET hive.exec.dynamic.partition.mode=nonstrict;
SET hive.exec.dynamic.partition=true;
SET hive.exec.max.dynamic.partitions.pernode=1000;
INSERT INTO TABLE user_rating_part PARTITION (year, month, day)
SELECT
  *,
  YEAR(last_modified),
  MONTH(last_modified),
  DAY(last_modified)
FROM
  user_rating"
```

如果数据已经导入到了新的目录中，那么我们只需要将这些目录添加成新的分区，通过执行以下操作完成。

```
#!/bin/bash
hive -e "ATLER TABLE user_rating_part
ADD PARTITION (year=2014, month=12, day=12)
LOCATION '/data/newdata'"
```

2. 合并/更新

在一些情况下，仅仅将新的事实记录添加到已有的表中是远远不够的。有时候我们实际上要对已有的信息进行更改，比如，我们有缓慢变化的维表。假定当用户修改信息（如职位）时，我们需要更新 user 数据集，还要更新 user_history 数据集的内容。在这一部分，

我们来关注如何将更新合并到 user 数据集上。

你应该还记得，HDFS 是一个一次写入（write-once）的文件系统。将数据存入 HDFS 后，我们不能通过修改文件的一小部分来更新一条记录。我们需要将整个已经存在的记录从文件中读取出来，然后将其复制到一个新的文件中，并将需要进行的更改应用到这个新的文件上。最后，我们更新 Hive 的元数据信息，将表指向更新后的新文件。

如果要更新的表非常大，读取非常耗时，执行更改也会很低效。这里有三种方式可以解决问题。

- 也许这张大表可以进行分区，修改操作只需涉及一小部分分区。举例来说，如果销售订单只能在 90 天之内取消，那么只有包含最新 90 天的分区才需要复制和更改。
- 要知道 Hadoop 工作起来并不总是快速的，但它的扩展性很好。如果你需要让复制和更改大表的操作加速，最简单的方案大概是添加集群的节点，保证所需的 I/O 和处理资源。
- 如果“缓慢更改”的维表的确很大，甚至更改也比较频繁，那么或许 HDFS 不是最适合的存储方式，可以考虑将频繁更新的表存储到 HBase 中。Hive 和 Impala 都支持对 HBase 和 HDFS 上的数据表进行关联。注意，在优化更新的同时，这样做的代价是减慢了某些查询的速度。从大批量扫描操作的角度来讲，这种在数据仓库中很常见的使用方式在 HBase 中比在 HDFS 中更为缓慢。

假定已经确定使用 HDFS 作为数据存储，而且对于这张用户信息表，我们需要将 Sqoop 导出的新数据更新到整个表中。在 Sqoop 的增量导入任务执行完毕之后，我们拥有了包含数据的以下两个目录。

- /data/movielens/user

  已存在的表。

- /etl/movielens/user_upserts

  新的记录。有一些记录是新增的，有一些则需要替换表中已存在的记录。upsert 一词融合了更新（update）和插入（insert）。在当前上下文中，该数据集包含的记录是我们需要进行更新和插入的。

user_upserts 数据的采集查询语句如下所示。

```
#!/bin/bash
sqoop job --create user_upserts_import --meta-connect \
jdbc:hsqldb:hsql://${SQOOP_METASTORE_HOST}:16000/sqoop \
--import --connect jdbc:mysql://mysql_server:3306/movielens \
--username myuser --password mypass \
-m 8 --incremental append --check-column last_modified --split-by last_modified \
--as-avrodatafile --query \
'SELECT
user.id,
user.age,
user.gender,
occupation.occupation,
zipcode,
last_modified
FROM user
```

```
JOIN occupation
ON (user.occupation_id = occupation.id)
WHERE ${CONDITIONS}' \
--target-dir /etl/movielens/user_upserts
```

然后，我们给这些需要 upsert 的记录创建一张 Hive 表，以提供结构化视图。

```
#!/bin/bash
hive -e \
"CREATE EXTERNAL TABLE IF NOT EXISTS user_upserts(
  id INT,
  age INT,
  occupation STRING,
  zipcode STRING,
  last_modified BIGINT)
STORED AS AVRO
LOCATION '/etl/movielens/user_upserts'"
```

此时，我们需要做的就是编写一条查询语句，将 user_upserts 与已存在的数据集进行合并，将合并的结果写入用户信息表。然而，在 Hive 中，`INSERT OVERWRITE` 操作是非原子性的。对于一个非原子性的插入，读取该表的用户可能会看到不一致的视图。因此，我们创建一张用户信息的临时表（称之为 user_tmp），将合并后的结果插入这张表，然后再原子性地修改用户信息表，使之接入新的数据。

如下是 user 和 user_tmp 的 `CREATE TABLE` 语句。

```
#!/bin/bash
hive -e \
"CREATE EXTERNAL TABLE IF NOT EXISTS user(
  id INT,
  age INT,
  occupation STRING,
  zipcode STRING,
  last_modified TIMESTAMP)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION '/data/movielens/user'"

DT=$(date "+%Y-%m-%d")
hive -e \
"CREATE EXTERNAL TABLE IF NOT EXISTS user_tmp(
  id INT,
  age INT,
  occupation STRING,
  zipcode STRING,
  last_modified TIMESTAMP)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION '/data/movielens/user_${DT}'"
```

注意，除了 LOCATION 子句指定的位置信息不同，user 和 user_tmp 的表定义是完全一致的。

接下来的目标是读取已存在的表，读取新增的记录，并创建二者合并后的表。新的表将包含全部已存在的记录（与新记录发生冲突的记录除外）以及所有新增的记录。

在这个例子中，表都是不带分区的。但是如果原来的用户信息表是带分区的，我们也需要对这张表按照相同的方式进行分区。注意，我们为这张表指定了一个位置信息，在这个位置包含了最新的数据。这样做能够存储用户信息表的历史数据，并且可以让更改前的数据很方便地复原。

接下来就是实际的合并操作。

```
#!/bin/bash
hive -e "
INSERT OVERWRITE TABLE user_tmp
  SELECT
    user.*
  FROM
    user
    LEFT OUTER JOIN
    user_upserts
    ON (user.id = user_upserts.id)
  WHERE
    user_upserts.id IS NULL
  UNION ALL
  SELECT
    id, age, occupation, zipcode, TIMESTAMP(last_modified)
  FROM
    user_upserts"
```

以上查询语句乍看有些复杂。它只是从用户信息表中拿到了所有不会被 user_upserts 记录替代的记录（两表关联，我们只选 user_upserts.id 为空的记录，这意味着对应的 ID 没有新的记录），然后再加上所有新的记录（UNION ALL SELECT * FROM USER_UPSERTS）。我们接下来将所有的结果插入 user_tmp 数据集中。

```
#!/bin/bash
hive -e "ALTER TABLE user SET LOCATION '/adta/movielens/user_${DT}'"
```

这样，用户信息表中就是当前位置的合并后的数据了。现在，我们可以删掉 user_upserts 表，或许几天之后先前的 user_${DT} 目录也可以删掉了。

10.6.4　数据聚合

很多情况下，在 Hadoop 上执行 ETL 操作的重要性在于，Hadoop 支持那些在关系型数据库中执行起来很耗资源的聚合操作。就复杂的聚合操作来讲，人们会认为这是一种高级的分析。在非常大的数据实体上进行极为简单的聚合操作（求和、计数或求平均）时，很多人不知道可以对这类操作进行效果显著的优化。这类聚合对于 Hadoop 来说，是天然高度并行的，是非常适合在 Hadoop 上进行的操作。Hadoop 的扩展性与相对低廉的成本为聚合 SLA 的操作提供了大量的资源。这一类的聚合包括电信运营商统计上线设备的错误计数、农业公司统计辖内每一寸土地的湿度平均值，等等。

对于这里讨论的情形，聚合的一个例子就是统计指定用户对特定电影的评分次数。如果涉及的事实表是一张大表，这就是一个非常庞大的查询。不过，这样的查询在大规模的 Hadoop 集群上执行速度是相当快的，这是因为用户 - 电影计数是能够并行独立计算的。

以上聚合在 Hive 或 Impala 中可以使用 `CREATE TABLE AS SELECT`（或 CTAS）模式完成。

```
#!/bin/bash
hive -e "
CREATE TABLE user_movie_count AS
SELECT
  movie_id,
  user_id,
  COUNT(*) AS count
FROM
  user_rating_fact
GROUP BY
  movie_id,
  user_id"
```

就我们的例子而言，一个更为有用的聚合是计算每部电影的平均得分。

```
#!/bin/bash
hive -e "
CREATE TABLE avg_movie_rating AS
SELECT
  movie_id,
  ROUND(AVG(rating), 1) AS rating
FROM
  user_rating_part
GROUP BY
  movie_id"
```

注意，上面只是以 Hive 或 Impala 举例说明，你可以采用任意 Hadoop 处理工具。某些聚合操作涉及特定算法，可能在 SQL 中无法实现（比如地理空间函数，geospatial function）或者用 SQL 表述过于复杂（如第 8 章提到的会话生成的例子）。要知道，ETL 系统那些数据变换只能通过 SQL 来完成，而 Hadoop 提供了大量的工具，可以用来解决任何问题。如果你选择使用其他的工具执行聚合操作，只需要在结果集上创建一张外表，用户就可以通过 Impala 对数据进行访问了。

或者，正如 10.6.5 节要讨论的那样，无论通过什么方式产生聚合后的数据集，我们只需要使用 Sqoop 将结果集导出到关系数据仓库中。到了这里，所有的 BI 工具都能施展身手。

10.6.5 数据导出

数据处理完毕之后，我们需要决定怎样使用这些结果。一种做法是简单地将数据保留在 Hadoop 中，使用 Impala 和集成的 BI 工具进行查询。Hive 也可以完成这种查询。然而，如果有低延迟访问的需求，Impala 会更为合适。

不过，很多情况下，很多业务在应用研发上投入了大笔资金。无论是自行研发的还是第三方的应用，它们均采用了已有的关系型数据仓库。将这些应用全部转入 Hadoop 是一大笔

投资，未必划算。

在这种情况下，我们需要以一种高效的方式将处理后的数据从 Hadoop 中导出，然后导入关系型数据仓库。这里有几种办法。你可以使用能够与 Hadoop 集成的 ETL 工具（如 Informatica 和 Pentaho）。一些数据库厂商（最为值得一提的就是 Oracle），拥有经过优化的连接器，可以从 Hadoop 向自身数据库导入数据。最流行的方式仍然是采用 Sqoop。

使用 Sqoop 将数据传输到关系型数据库是个很直接的过程。这时执行的不是 `sqoop import`，而是 `sqoop export` 命令。接下来批量插入行并分批次将数据提交给数据库。这就意味着，数据导出的过程会有多次数据库提交。如果导出中途出错，已经导入数据库的那部分数据就需要清理，而这处理起来令人头疼。因此，我们推荐对 Sqoop 进行设置。在导入数据库时先导入到暂存表中，当所有的数据全部导出完成后，再将数据移入实际要导入的表。

值得注意的另外一点是，导出数据可能会涉及记录行的插入和更新。Sqoop 支持三种模式：仅插入、仅更新和混合。你要保证为自己的应用场景选择正确的导出模式。更新需要一个更新的键，这个键应该是表的唯一键。Sqoop 会根据该键识别记录是否已存在，进而执行更新。因此在 Hadoop 上和在关系型数据仓库中，这个键必须是相同的。

以下是使用 Sqoop 导出到一个暂存表的例子，插入和更新均包括在内。

```
#!/bin/bash
# 执行以下命令前,DWH中的目的表应当是存在的,
# 并且对于运行Sqoop任务的用户来讲,用于写操作权限
# 类似的建表语句如下:
# CREATE TABLE avg_movie_rating(movie_id INT, rating DOUBLE);

sqoop export --connect \
jdbc:mysql:/mysql_server:3306/movie_dwh --username myuser --password mypass \
--table avg_movie_rating --export-dir /user/hive/warehouse/avg_movie_rating \
-m 16 --update-key movie_id --update-mode allowinsert \
--input-fields-terminated-by '\001' --lines-terminated-by '\n'
```

最后提一条建议：如果你的数据库有 Sqoop 优化后的连接器，那么就使用它。这些连接器支持批量加载，减少了数据库的加锁，显著地提高了导出性能。

10.6.6 流程调度

整个 ETL 处理过程开发完毕之后，就到了让工作流自动化的时候了。如之前章节讨论的那样，流程调度可以通过 Oozie（如果要使用开源解决方案的话）或其他已有的流程调度工具来完成，Autosys、Activebatch 以及 UC4 都是常用的工具。

如果决定使用 Oozie，那么工作流应该包含以下四点。

- 一个用于获取数据的 Sqoop 操作。
- 一个用于执行数据集关联、分区、合并的 Hive 操作。
- 多个用于完成聚合的 Hive 或 MapReduce 操作。
- 一个用于将结果导出至数据仓库的 Sqoop 操作。

在点击流分析的案例中，我们看到了 Hive 和 MapReduce 操作的例子。在这里，让我们简单看一下 Sqoop 操作。

```
<action name="import_facts">
   <sqoop xmlns="uri:oozie:sqoop-action:0.4">
     <job-tracker>${jobTracker}</job-tracker>
     <name-node>${nameNode}</name-node>
     <command>job -exec user_rating_import \
     --meta-connect jdbc:hsqldb:hsql://edgenode:16000/sqoop</command>
     <archive>/tmp/mysql-connector-java.jar#mysql-connector-java.jar</archive>
     <file>/tmp/hive-site.xml#hive-site.xml</file>
   </sqoop>
   <ok to="end"/>
  <error to="kill"/>
 </action>
```

注意，要执行存储在 Sqoop 元数据服务中的一个任务，我们只需告诉 Sqoop 要执行哪一个任务（这里需要确保集群所有的节点均可以连接 Sqoop 的元数据）。我们通过 <command> 参数来完成该项指定。我们还需要告知 Sqoop 和 Oozie，JDBC 驱动的位置以及 Hive 配置文件的位置，这两点分别通过 <archive> 和 <file> 参数进行指定。

注意，我们的工作流将会包含多个 Sqoop 动作（每个表一个）、多个分区或合并的步骤、多个聚合操作，等等。我们可以通过第 6 章描述的 fork-and-join 模式实现。

图 10-8 展示的就是工作流程整体的样子。

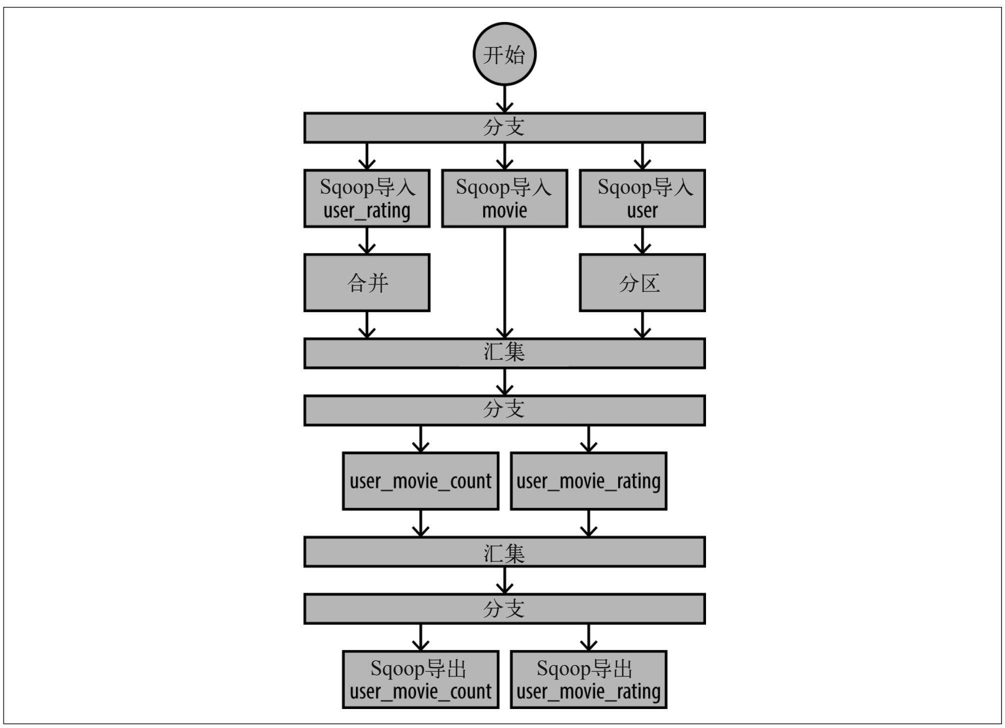


图 10-8：ETL 处理的 Oozie 工作流

10.7 小结

本章研究了在已有企业级数据仓库上部署 Hadoop 的难点。我们了解到，使用 Hadoop 作为 EDW 的补充满足了 ETL 任务的 SLA，减少了对 EDW 资源的过多占用，把 Hadoop 打造成了一个高精度数据的在线归档服务和一个支持探索性分析的绝佳数据平台。我们以 MovieLens 为例介绍了 OLTP 系统中的数据集，讲解了数据如何导入 Hadoop，以及在 Hadoop 中可以对数据进行怎样的转换。接下来，我们基于 Hadoop 的 MovieLens 数据集，生成了一些聚合表，并将其导出至数据仓库中。我们还展示了如何使用 Oozie 这样的调度工具协调所有的导入、处理以及导出。你也可以将半结构化或非结构化数据与 EDW 中的传统结构化数据关联。

希望通过本章的学习，你可以使用 Hadoop 补充已有的 EDW，打破原有架构的限制，从数据中汲取更多价值。

附录 A
Impala中的关联

第 3 章就 Impala 的基本情况和基本原理进行了介绍。在本附录中，我们来了解一下 Impala 如何计划并执行分布式关联联查询（join query）。在本书写作之时，Impala 支持两种关联策略：广播式关联（broadcast join）与分区后散列关联（partitioned hash join）。

A.1 广播式关联

Impala 最先支持的关联就是广播式关联，这也是默认的关联模式。在进行广播式关联时，Impala 会读取较小的数据集，并将其分发给所有该查询计划涉及的 Impala 后台程序[1]。参与执行该计划的 impalad 接收到数据集之后，就会将其以哈希表（hash table），[2] 的形式放置于内存中。接下来，每个 impalad 会就近读取较大数据集的数据，并参照内存中的哈希表，就每一行数据进行比对，查看是否匹配（即进行散列关联）。由于不必将较大数据集的全部内容放到内存中，因此 Impala 使用 1 GB 的缓冲区来分段读较大的表，然后分别依次执行关联操作。

图 A-1 和图 A-2 是工作原理图解。图 A-1 展示了每个 impalad 如何缓存较小数据集。该关联策略较为简单，但是需要关联的二者之中至少有一个是较小的表。

关于广播式关联，有以下几点需要注意。

- 在每个节点上，较小的数据集都会占用相应的内存。假设你有三个节点，每个节点上的 Impala 配置有 50 GB 内存，对于广播式关联，较小数据集的上限略高于 40 GB，而集群的总内存有 150 GB。

注 1：由于实际运行时该进程名为 impalad，以下均称之为 impalad。——译者注

注 2：hash 一词可以是哈希或散列，本文依语境替换使用。——译者注

- 内存缓冲区的数据不是整个小表的全部数据，而是列的哈希表——要关联的列以及查询时涉及的列。如果一次查询仅涉及小表的一小部分列，那么只有这部分列的数据会占用相应的内存。
- 小表的数据会分发到每一个 impalad。这就意味着，数据分发会经由网络，导致查询的吞吐量受限于集群带宽。
- Impala 使用基于代价的优化器（cost-based optimizer）评估表的大小，判断哪一张表较小以及对应的哈希表会需要多少内存，从而决定是否进行广播式关联。如果基于代价的优化器能够了解查询中各个表的大小，它就可以生成更为合理的查询计划。因此，定期收集和计算数据的统计信息对 Impala 的性能提升很有帮助。

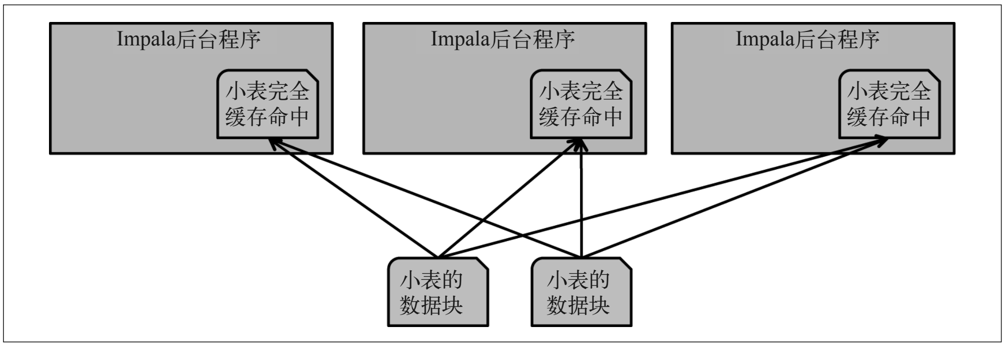


图 A-1：广播式关联中的小表缓存

待数据广播和缓存后，较大数据集的数据就会以流式的方式与内存中较小数据集生成的哈希表进行比对。每一个 impalad 会处理较大数据集的一部分，首先扫描数据到内存，然后传给散列关联方法，如图 A-2 所示。

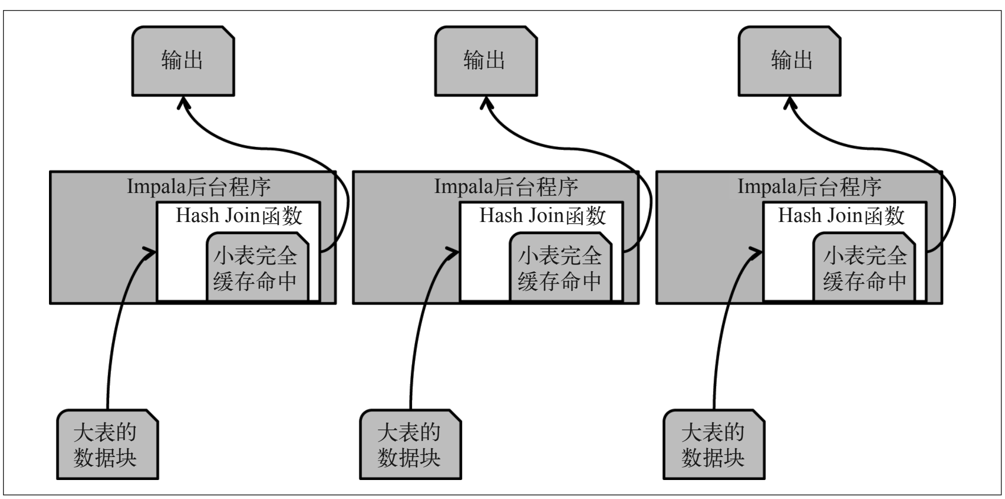


图 A-2：Impala 中的广播式关联

通常情况下，每个 impalad 会尽量从本地磁盘读取构成较大数据集的数据，以减少对网络的使用。此时较小数据集已缓存在每个节点中，该阶段已无网络传输开销。不过，如果关联后的结果集需要参与查询计划后续的其他关联操作，这里也会发生额外的网络传输。图 A-2 假定只有单个关联操作发生。

A.2 分区后散列关联

分区后散列关联会产生更多网络活动，不过大数据集可进行关联，而不必有一整张表放置于单个节点。如果统计数据表明该表数据过多，无法放置于单个节点内存中，或者一个查询中添加了提示，要进行分区后散列关联操作，该关联策略就会使用。

当执行分区后散列关联操作（hash join，也称 shuffle join）时，每一个 impalad 会读二者中较小的表的本地数据，使用一个散列函数对其进行分区，然后将每个分区发送到不同的 impalad 中。

如图 A-3 所示，分区后散列关联与广播式关联不同。前者每一个节点缓存数据集的一个子集，而后者的每一个节点均存储整个数据集。由此，你可以更为高效地使用集群的内存资源。

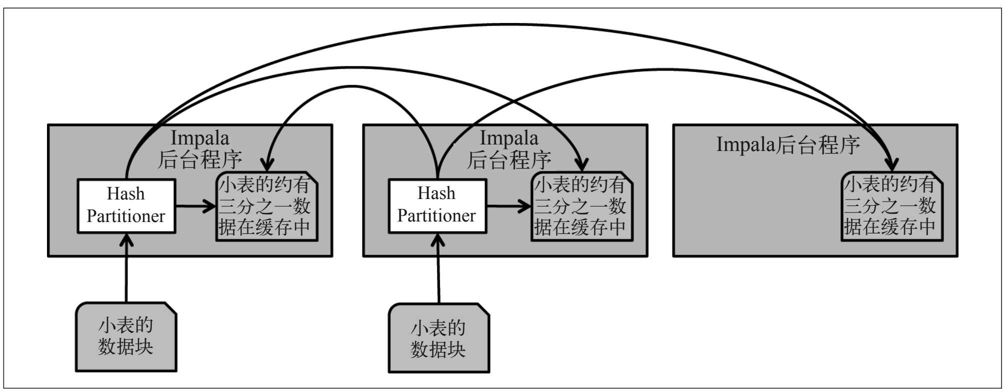


图 A-3：在分区后散列关联中，小表的数据缓存图解

图 A-4 展示了分区后散列关联的具体执行过程。图中有三个 impalad，为了便于解释，我们只关注其中的一个 impalad。但你要知道，所有的 impalad 都会执行类似的操作。

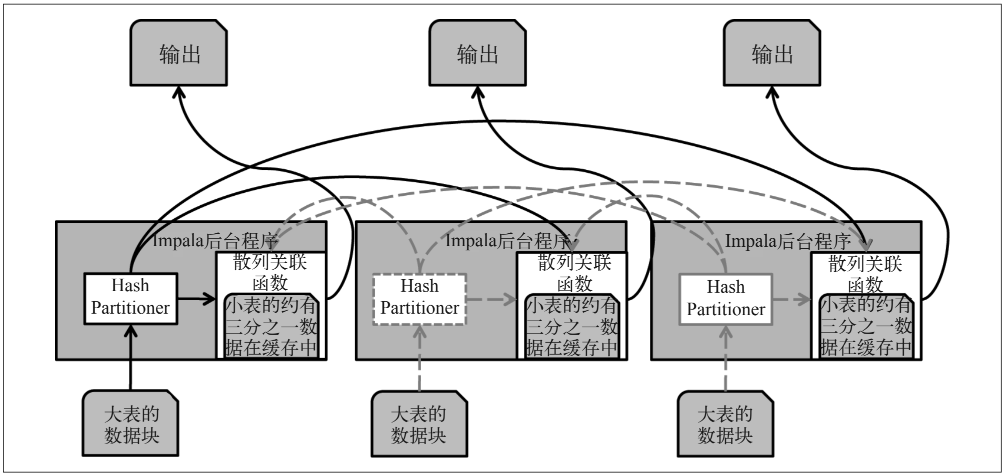


图 A-4：Impala 中的分区后散列关联

如图所示，较大的表也会使用相同的散列函数形成不同的分区，将数据发送到对应的节点，在各个节点上执行关联操作，与小表的数据进行匹配过滤。注意，广播式关联只在网络上发生一次数据分发。而进行分区后散列关联时，两个表的数据会在多个节点间分发，网络传输开销更大。

总结一下，Impala 有两种关联策略：广播式关联与分区后散列关联。前者需要相对较多的内存，只适合存在小表的应用场景，后者则会占用较多的网络资源，查询会慢一点，但是适用于大表的应用场景。

Impala 支持在 SQL 语句中添加提示 `[BROADCAST]` 或 `[SHUFFLE]`，以明确选用哪种关联策略。代码示例如下。

```
select
from foo f join [BROADCAST] bar b
on f.bar_id = b.bar_id
```

但是，我们不推荐在 SQL 中使用提示，因为这会导致将关联策略硬编码到查询语句中。一张表也许最初是小表，但其数据量可能会增大，这样一来使用 `[BROADCAST]` 提示就不再合适了。Impala 查询出错的一个常见的原因就是广播式关联将过大的表进行分发，导致内存不足。这种情况应该使用分区后散列关联。

一个更好的做法是允许 Impala 自行选择关联策略。要做到这一点，就需要收集数据的统计信息。通过以下命令可以实现该目标（以下语句 Impala 语法不支持，应当是 COMPUTE ，译者注）。

```
ANALYZE TABLE <Ttable> COMPUTE STATS <Ttable>;
```

该指令可以触发 Impala 的扫描操作，并就数据量和数据分布进行统计层面的信息收集。从而允许 Impala 自行选择关联策略，以优化查询性能。如果一张表有超过 10% 的内容发生了更新，则推荐对该表执行以上指令。

作者简介

Mark Grover Apache Sentry 项目管理委员会成员，《Hive 编程指南》作者之一，曾参与 Apache Hadoop、Apache Hive、Apache Sqoop 以及 Apache Flume 等项目，并为 Apache Bigtop 项目和 Apache Sentry（项目孵化中）项目贡献代码。

Ted Malaska Cloudera 公司的资深解决方案架构师，致力于帮助客户更好地掌握 Hadoop 及其生态系统。曾任美国金融业监管局首席架构师，指导建设了包括网络应用、服务型架构以及大数据应用在内的大量解决方案。曾为 Apache Flume、Apache Avro、YARN 以及 Apache Pig 等项目贡献代码。

Jonathan Seidman Cloudera 公司的解决方案架构师，协助合作伙伴将的解决方案集成到 Cloudera 的软件栈中。芝加哥 Hadoop 用户组及芝加哥大数据的联合创始人、《Hadoop 实战》技术编辑。曾任 Orbiz Worldwide 公司大数据团队技术主管，为最为繁忙的站点管理了承载海量数据的 Hadoop 集群。也曾多次在 Hadoop 及大数据专业会议上发言。

Gwen Shapira Cloudera 公司的解决方案架构师，知名博主，拥有 15 年从业经验，协助客户设计高扩展性的数据架构。曾任 Pythian 高级顾问、Oracle ACE 主管以及 NoCOUG 董事会成员，活跃于诸多业内会议。

封面介绍

本书封面动物为海牛（海牛科），现存的三种海牛分别是：亚马孙海牛、西印度海牛与西非海牛。

海牛是纯粹的水生哺乳动物，重达 1300 磅。这个名字来源于泰诺（Taino）语，意为“乳房”。人们认为这种动物的祖先是 6000 万多年前的四足陆生哺乳动物，与大象和岩狸是近亲。

虽然只生活在水下，但海牛通常长有粗糙的毛发和胡须。它们厚厚的皮肤布满皱纹，卷曲的上唇可以用来收集食物。海牛是食草动物，大部分时间都在觅食淡水中或海水中的植物。它们尤其喜欢漂浮的风信子、梭鱼草、水浮莲和红树叶。海牛的上唇可以分成左右两片，二者可以独立运动，在咀嚼进食时，嘴唇甚至可以将植物撕开。

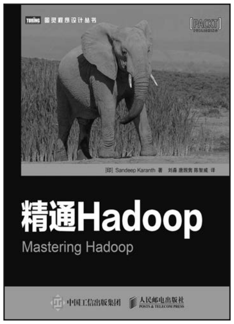
海牛通常独居，寻找配偶或护理幼体的时期除外。它们会发出各种各样的声音进行交流，这与海豚和海豹的沟通方式类似。海牛能够从事较为复杂的相关性学习，执行简单的任务。

海牛在野外没有天敌，它们最大的威胁来自人类——船只碰撞、水质污染以及栖息地破坏。船只对海牛的伤害非常普遍。在 2008 年，大西洋中部有四分之一的海牛因此丧生。此外，还有很多海牛严重受伤，留下疤痕。三种海牛均已列入世界自然保护联盟（World Conservation Union）的濒临灭绝动物名单。

登上 O’Reilly 图书封面的许多动物都已濒临灭绝。它们在这世上弥足珍贵。要知道如何为保护动物贡献自己的一份力量，请访问 animals.oreilly.com。

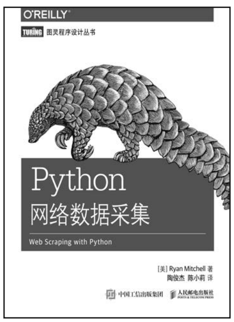
封面图片来自《布罗克豪斯百科全书》（*Brockhaus Lexicon*）。

延 展 阅 读



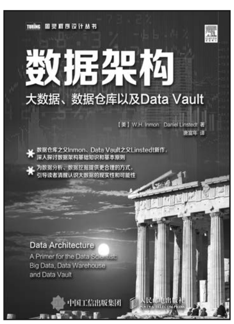
本书是一本循序渐进的指导手册，重点介绍了Hadoop的高级概念和特性。内容涵盖了Hadoop 2.X版的改进，MapReduce、Pig和Hive等的优化及其高级特性，Hadoop 2.0的专属特性（如YARN和HDFS联合），以及如何使用Hadoop 2.0版本扩展Hadoop的能力。

书号：978-7-115-41105-1
定价：49.00 元



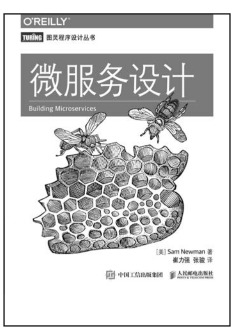
网络上的数据量越来越大，单靠浏览网页获取信息越来越困难，如何有效地提取并利用信息已成为一个巨大的挑战。本书采用简洁强大的Python语言，全面介绍网络数据采集技术，教你从不同形式的网络资源中自由地获取数据。

书号：978-7-115-41629-2
定价：59.00 元



本书是数据仓库之父Inmon的新作，探讨数据的架构和如何在现有系统中最有效地利用数据。本书的主题涵盖企业数据、大数据、数据仓库、Data Vault、业务系统和架构。主要内容包括：在分析和大数据之间建立关联，如何利用现有信息系统，如何导出重复型数据和非重复型数据，大数据以及使用大数据的商业价值，等等。

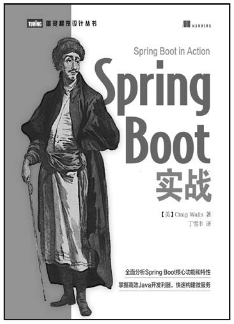
书号：978-7-115-43843-0
定价：69.00 元



本书全面介绍了微服务的建模、集成、测试、部署和监控，通过一个虚构的公司讲解了如何建立微服务架构。主要内容包括认识微服务在保证系统设计与组织目标统一上的重要性，学会把服务集成到已有系统中，采用递增手段拆分单块大型应用，通过持续集成部署微服务，等等。

书号：978-7-115-42026-8
定价：69.00 元

延 展 阅 读



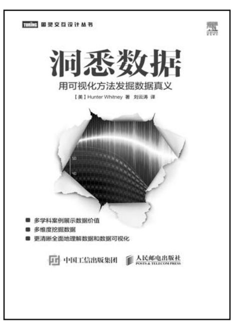
- 掌握自动配置和起步依赖，学会用很少的显示配置构建完整的Spring应用程序
- 了解如何为Spring Boot应用程序编写自动化集成测试
- 开发Spring Boot CLI应用程序
- 探秘Actuator的Web端点、远程shell和JMX MBean
- 自如部署各种Spring Boot应用程序

书号：978-7-115-43314-5
定价：59.00 元



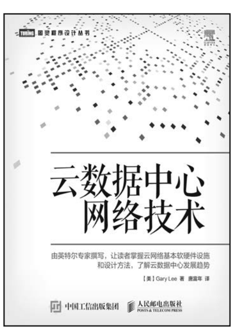
本书作者基于自身的工作经验，结合具体的实例，以轻松的语气探讨了团队成功的关键因素，并阐述了如何克服团队中的工程和人际问题，让团队成员可以将更多精力和时间投入产品创造中。

书号：978-7-115-43418-0
定价：45.00 元



本书为了解数据可视化的重要内容和功能提供了多学科的视角，通过各种各样的案例分析，来演示可视化如何让数据变得更清晰、更全面，通过对数据可视化的广泛用途和适用性的讨论，来了解它如何让数据变得更加让人容易接受和理解。

书号：978-7-115-41470-0
定价：69.00 元



- 云网络背景概述，阐述了原有网络技术如何向分布式、基于云的网络发展
- 剖析云网络关键组件：交换结构技术，拓扑结构，网络标准
- 了解架构的发展、高性能计算、大数据分析等，展望云网络数据中心的未来
- 展示英特尔公司网络团队的前沿交换结构技术细节

书号：978-7-115-40518-0
定价：49.00 元

O'REILLY®

Hadoop应用架构

本书就使用Apache Hadoop端到端数据管理方案提供专业架构指导。其他书籍大多针对Hadoop生态系统中的软件，讲解较为单一的使用方法，而本书偏重实践，在架构的高度详细阐释诸多工具如何相互配合，搭建出打磨之后的完整应用。书中提供了诸多案例，易于理解，配有代码解析，知识点一目了然。

为加强训练，本书后半部分内容深入讲解案例，涵盖最为常见的Hadoop应用架构。无论是设计Hadoop应用，还是将Hadoop同现有数据基础架构集成，本书都可以提供详实的参考。

- **使用Hadoop进行数据存储和建模的着眼点和思路**
- **将数据输入、输出系统的最佳方案**
- **MapReduce、Spark和Hive等数据处理框架介绍**
- **数据去重、窗口分析等常见Hadoop处理模式应用**
- **在Hadoop上采用Giraph、GraphX等图形处理工具**
- **综合使用工作流以及Apache Oozie等调度工具**
- **以Apache Oozie、Apache Spark Streaming和Apache Flume进行近实时流处理**
- **点击流分析、欺诈检验和数据仓库的架构案例**

Mark Grover
Apache Sentry项目管理委员会成员，Hadoop等多个Apache项目的代码贡献者，《Hive编程指南》作者之一。

Ted Malaska
Cloudera资深解决方案架构师，致力于帮助客户更好地掌握Hadoop及其生态系统。

Jonathan Seidman
Cloudera解决方案架构师，协助合作伙伴将解决方案集成到Cloudera的软件栈中。

Gwen Shapira
Cloudera解决方案架构师，知名博主，拥有15年从业经验，协助客户设计高扩展性的数据架构。

DATABASES

封面设计：Ellie Volckhausen　**马冬燕**

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／数据库

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-44243-7

ISBN 978-7-115-44243-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks